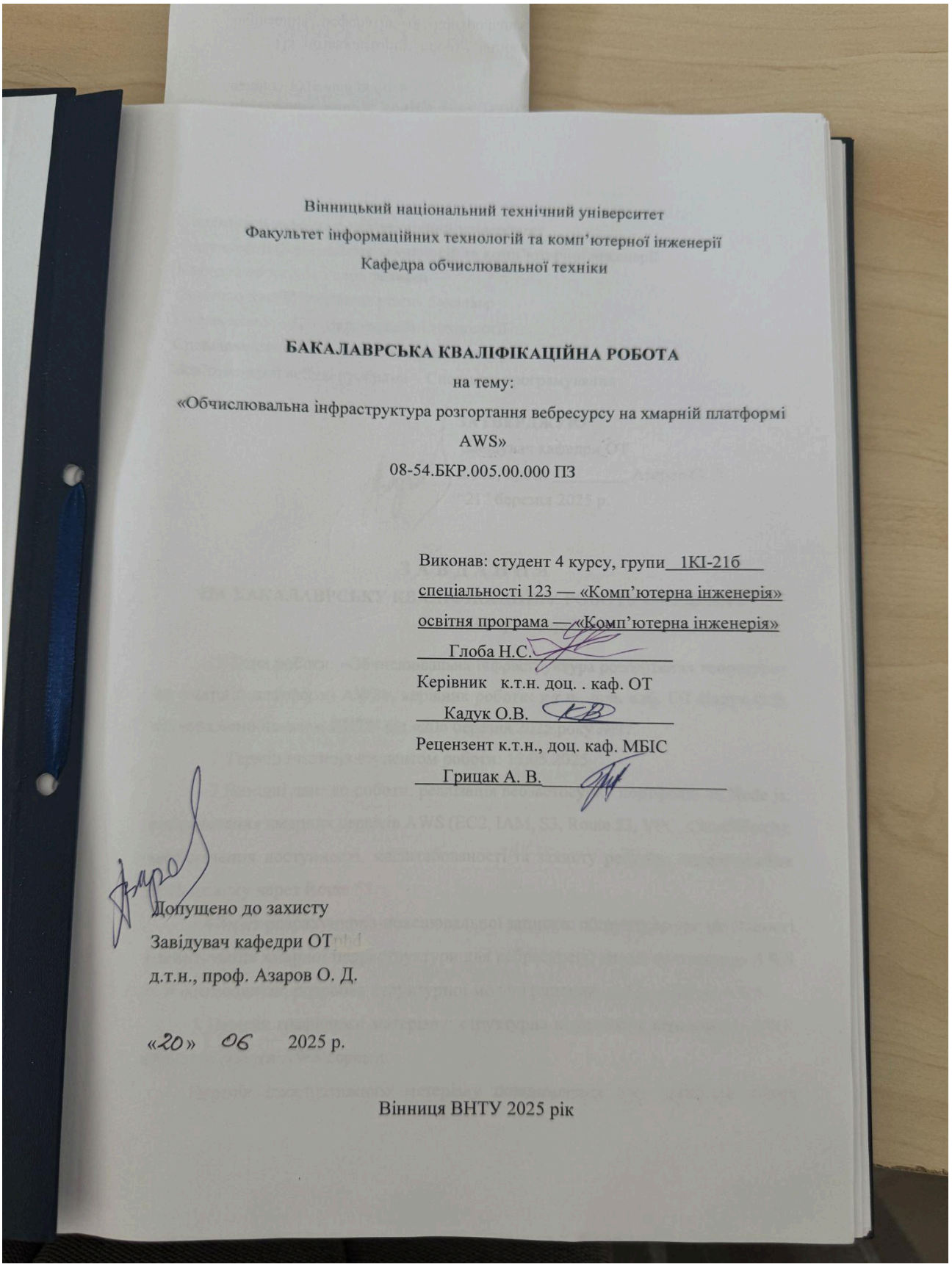

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Обчислювальна інфраструктура розгортання вебресурсу на хмарній платформі AWS»

08-54.БКР.005.00.000 ПЗ

Виконав: студент 4 курсу, групи 1КІ-21б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма — «Комп'ютерна інженерія»
Глоба Н.С.
Керівник к.т.н. доц. . каф. ОТ
Кадук О.В.
Рецензент к.т.н., доц. каф. МБІС
Грицак А. В.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.

«20» 06 2025 р.

Вінниця ВНТУ 2025 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо-кваліфікаційний рівень бакалавр
Галузь знань – 12 – Інформаційні технології
Спеціальність – 123 – Комп'ютерна інженерія
Освітньо-професійна програма – Системне програмування

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. _____ Азаров О. Д
"21" березня 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Глобі Назарію Сергійовичу

1 Тема роботи: «Обчислювальна інфраструктура розгортання вебресурсу на хмарній платформі AWS», керівник роботи: к.т.н., доц. каф. ОТ Кадук О.В, затверджено наказом ВНТУ від «20» березня 2025 року №97.

2 Термін подання студентом роботи: 13.05.2025

3 Вихідні дані до роботи: реалізація вебзастосунку портфоліо на Node.js; використання хмарних сервісів AWS (EC2, IAM, S3, Route 53, VPC, CloudWatch); забезпечення доступності, масштабованості та захисту ресурсу; налаштування DNS-домену через Route 53.

4 Зміст розрахунково-пояснювальної записки: обгрунтування доцільності використання хмарної інфраструктури для вебресурсів; аналіз архітектури AWS та її компонентів; розробка структурної моделі рішення; вибір сервісів AWS.

5 Перелік графічного матеріалу: структурна візуалізація середовища AWS, алгоритм роботи AWS сервісів.

Перелік ілюстративного матеріалу повідомлення про успішний запуск

інстансу.

6 Консультанти розділів роботи наведені у таблиці 1 .

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	завдання прийняв
1-4	Кадук О.В., доцент кафедри ОТ	21.03.2025 р.	13.05.2025 р.
Нормконтроль	ас. Каф. ОТ Швець С.І	13.05.2026 р.	30.05.2025 р.

7 Дата видачі завдання «21» березня 2025 р.

8 Календарний план приведений в таблиці 2

Таблиця 2 — Календарний план

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів	Підпис
1	Постановка задачі рооти	21.03.2025	
2	Пошук матеріалів та інформаційних джерел	21.03.25— 30.03.25	
3	Аналіз існуючих хмарних технологій	31.03.25— 25.04.25	
4	Розгортання і тестування веб додатку	26.04.25— 02.05.25	
5	Оформлення пояснювальної записки та ілюстрованого матеріалу	03.05.25— 11.05.25	
6	Перевірка якості виконання бакалаврської кваліфікаційної роботи та усунення недоліків	13.05.25	

Студент _____ Назарій ГЛОБА

Керівник роботи _____ к.т.н. доц. Олександр КАДУ

АНОТАЦІЯ

УДК 621.3

Глоба Н.С. Обчислювальна інфраструктура розгортання вебресурсу на хмарній платформі AWS. Бакалаврський кваліфікаційна робота зі спеціальності 123 «Комп'ютерна інженерія», освітня програма «Комп'ютерна інженерія». Вінниця: ВНТУ, 2025, 95 с.

На укр. мові. Бібліогр.: 34 назв; рис.: 20; табл. 1.

В бакалаврській кваліфікаційній роботі створено обчислювальну інфраструктуру для розгортання Node.js вебпортфоліо на хмарній платформі Amazon Web Services (AWS). Розглянуто використання сервісів AWS: EC2 для віртуальних серверів, S3 для зберігання контенту, IAM для керування доступом, Route 53 для налаштування доменного імені, CloudWatch для моніторингу та VPC для побудови ізольованої мережі. Здійснено практичне розгортання вебпортфоліо, налаштовано доменне ім'я, забезпечено заходи безпеки та організовано моніторинг системи. Проведено оптимізацію конфігурації ресурсів для підвищення ефективності та надійності роботи вебресурсу.

Ключові слова: обчислювальна інфраструктура, розгортання вебресурсу, Amazon Web Services, Node.js, EC2, безпека, моніторинг.

ABSTRACT

Globa N. S. Computational Infrastructure for Deploying a Web Resource on the AWS Cloud Platform.

Bachelor's qualification work in specialty 123 “Computer Engineering”, educational program “Computer Engineering”.

Vinnytsia: VNTU, 2025, 95 p.

In Ukrainian. Bibliography: 34 titles; Figures: 20; Tables: 4.

This bachelor's qualification work presents the creation of a computational infrastructure for deploying a Node.js web portfolio on the Amazon Web Services (AWS) cloud platform. The use of AWS services is considered: EC2 for virtual servers, S3 for content storage, IAM for access management, Route 53 for domain name configuration, CloudWatch for monitoring, and VPC for building an isolated network. A practical deployment of the web portfolio was carried out, the domain name was configured, security measures were implemented, and system monitoring was organized. The configuration of resources was optimized to increase the efficiency and reliability of the web resource.

Keywords: computational infrastructure, web resource deployment, Amazon Web Services.

ЗМІСТ

ВСТУП

Актуальність хмарних обчислень у сучасному світі важко переоцінити, вони відіграють важливу роль у багатьох сферах діяльності, зокрема в бізнесі, телекомунікаціях, Інтернеті, освіті та мультимедійних технологіях. Зі стрімким зростанням популярності хмарних сервісів та розвитку IoT усе більше уваги приділяється ефективному розгортанню вебресурсів у хмарному середовищі. Amazon Web Services (AWS) сьогодні є одним з лідерів серед таких платформ, надаючи широкий спектр сервісів, що допомагають організаціям швидше впроваджувати інновації, знижувати витрати та підвищувати продуктивність застосунків. Системи хмарного розгортання мають відповідати вимогам до продуктивності, масштабованості, надійності та безпеки, що робить дослідження їх розробки актуальним і затребуваним. Актуальність теми полягає в інтеграції вебресурсу у хмарну платформу AWS з метою розширення функціональних можливостей та масштабованості сервісу. Йдеться про створення інфраструктури, яка дозволить розгорнути веб-застосунок у хмарному середовищі, забезпечивши його надійну та масштабовану роботу. Така система має включати автоматизоване налаштування обчислювальних ресурсів, конфігурацію мережевих компонентів і сховищ даних, а також можливість безперервного оновлення та моніторингу застосунку.

Метою роботи є створення обчислювальної інфраструктури для розгортання вебресурсу на базі хмарної платформи AWS, яка забезпечуватиме двосторонній обмін даними з користувачами у режимі реального часу через веб-протоколи (HTTP/HTTPS), передачу файлів та даних застосунку через хмарні сервіси (TCP/IP) та керування ресурсами через веб-інтерфейс (консоль AWS або API).

Об’єктом дослідження є процеси розгортання, масштабування та керування вебресурсом у хмарній інфраструктурі AWS.

Предмет дослідження — апаратно-програмні засоби реалізації розгортання веб-застосунку на базі AWS.

Для досягнення мети були поставлені наступні **завдання**:

— а наліз проблематики реалізації систем розгортання вебресурсів у хмарі, зокрема питань продуктивності, затримок мережі та обмежених ресурсів;

— дослідження архітектури обміну даними між клієнтським застосунком та хмарним середовищем AWS через веб-протоколи;

— розробка функціональності для розгортання та безперервної роботи веб-застосунку в режимі реального часу з використанням хмарних сервісів AWS;

— реалізація функціоналу збереження та отримання файлів і даних вебресурсу з використанням хмарних сховищ AWS (наприклад, Amazon S3 та баз даних);

— організація керування конфігурацією та режимами роботи розгорнутого вебресурсу через зручний інтерфейс (AWS Management Console або спеціальний веб-застосунок адміністрування);

— тестування розробленої системи у реальних умовах та аналіз її стабільності, швидкодії та масштабованості, Node.js, EC2, security, monitoring.

Апробація результатів роботи здійснена у доповіді на Міжнародній науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2025)»

1 ОГЛЯД ТА АНАЛІЗ ТЕХНОЛОГІЙ ХМАРНИХ ОБЧИСЛЕНЬ

1.1 Основна концепція хмарних обчислень

У 21 столітті, без хмарних обчисоень не працює жодна IT-компанія, це технологія, яка повністю змінила інтерактивний світ. Замість того щоб купувати, налаштовувати, підтримувати, оновлювати власні сервери та сховища, можна придбати підписку у хмарному середовищі, що дозволяє компаніям зосередитись на розробці нових технологій, замість вісної підтримки власної інфраструктури.

Хмарні серидовища класифікуються за моделлю розгортання, організацєю інфраструктури та типом доступності. Розпочнемо із типу доступності (рис. 1.1), існують приватні, публічні, гібридні й галузеві хмари. Приватні хмари використовуються для використання однією організацією — ресурси хмари відносяться та керуються самою компанією. Хмара може розміщатись як на власних мацданчиках, так і в сторонніх дата-центрах провайдерів. Повною відмінністю являються публічні хмари, вони пропонуються провайдером для загального доступу та користуванню необмеженому колу користувачів. Доступ до ресурсів надається через інтернет, а інфраструктура використовується відразу багатьма клієнтами. Поєднанням двох, або більше, середовищ, зо залишається окремими сутностями, але пов’язані між собою спільними стандартами, або технологіями, є гібридна хмара. Даний підхід забезпечує портативність даних і додатків між середдовищами. Якщо прикладово, то клієнтська компанія може частину навантаження тримати у власному приватному дата-центрі, але якщо навантаження буде наближатись до пікового, то динамічно задіювати ресурси публчіної хмари, також така схема роботи називається cloud bursting. Також, окремо виділяють хмару спільноти — це інфраструктура, яку спільно використовує певна група організацій зі спільними інтересами, або вимогами. Такі сервіси менш поширені, але затребуані у сферах із жорсткими нормативними вимогами, де відразу кілька організацій об’єднуються для

створення спільної інфраструктури. Дані концепції — це основа хмарних обчислень, вони визначають якти чтном сервіси експлуатуються.

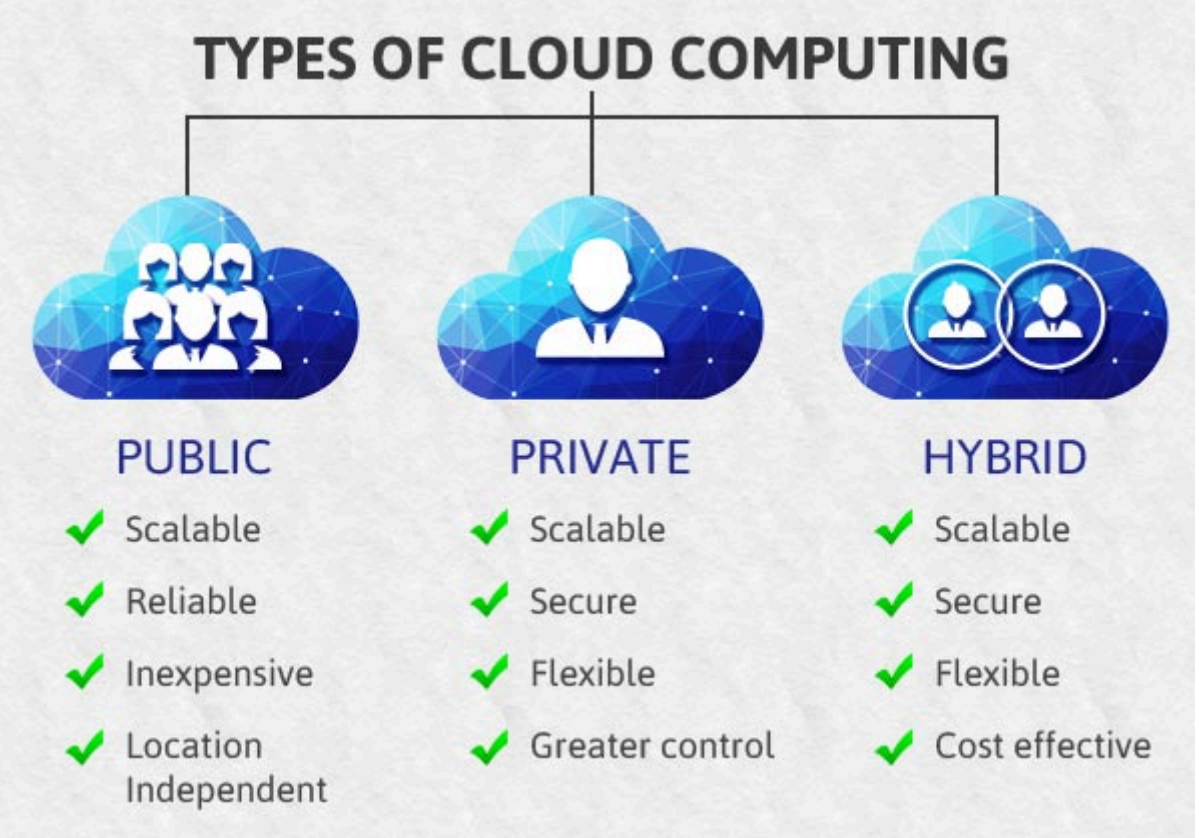

Рисунок 1.1 — Типи хмарних обчислень

Переважно, обчислення робляться за рахунок хмарних ресурсів — провайдери створюють із фізичних сервісів множину віртуальних, ізольованих один від одного, що й дозволяє ефективно ділити ресукрси між багатьма користувачами. В роботі, клієнти можуть комбінувати різні моделі: використовувати декілька сервісів SaaS (Software as a Servise, або програмне забезпечення як послуга), розробляти власні додатки на PaaS (Platform as a Service) та розгортати частину систем на IaaS (Infrastructure as a Service), також поєднувати піблічні та приватні хмари, отримуючи гібридні.

Багато сучасних підприємств дотримуються саме гібридної стратегії, оюираючи оптимальне середовище для окремо кожного завдання. Такий підхід дозволяє збалансувати вимоги бепеки, продуктивності та вартості, використовуючи переваги різних типів хмарних рішень та технологій.

1.2 Переваги та недоліки хмарних обчислень

Масовий перехід до хмарних обчислень, значно вплинув на організацію IT-галузі, та приніс багато сучасних переваг, в той же час, використання хмари наштовхує на певний ризик. Далі розглянемо головні переваги та недоліки хмарних обчислень для бізнесу.

Одною з головних переваг хмарних обчислень являється швидкість розгортання ресурсу та скорочення часу виходу на ринок. Провайдер надає можливості цілодобвого та миттєвого доступу до ресурсів і конфігурувати нові сервіси за хвилини. Клієнти можуть без проблем створювати тестові або релізнуті середовища, що значно прискорює цикл розробки продуктів. Також, зникає проблема в закупівлі та очікуванні до встановлення нового обладнання, для покращення роботи сервісу клієнта. Команди мають можливість в оеальному часі експерементувати з ідеями і відразу бачити результат.

Хмарні ресурси легко масштабуются вгору чи вниз залежно від поточного навантаження. Компанія може динамічно збільшити обчислювальну потужність або обсяг сховища у періоди пікового попиту і так само зменшити їх, коли потреба спадає. Це усуває проблему, коли потрібно інвестувати в обладнання, здатне витримати максимум навантаження, яке більшу частину часу простоює. У хмарі платформа автоматично підлаштовується під потреби бізнесу, забезпечуючи високу гнучкість у використанні IT-ресурсів.

Ще однією перевагою являється економія завдяки переходу від капітальних витрат (CapEx) до операційних (OpEx). Компанії сплачують тільки за ті ресурси, які фактично спожили, що допомагає уникнути переплат за невикористані потужності. Відпадає необхідність утримувати власні дата-центри, витрачатися на їхнє охолодження, електроживлення, персонал для підтримки тощо. За рахунок масштабування під реальні потреби бізнес може знизити загальні IT-витрати, а вивільнені кошти спрямувати на розвиток основної діяльності. Більше того, велика конкуренція між хмарними

провайдерами стимулює їх пропонувати все більш вигідні тарифи і знижки для клієнтів.

Дані та застосунки, розміщені в хмарі, доступні з будь-якої точки світу, де є Інтернет-з’єднання. Це полегшує віддалену роботу і співпрацю: працівники можуть одночасно працювати з документами чи бізнес-системами, незалежно від свого місцезнаходження. Хмарні сховища та сервіси спільної роботи (наприклад, офісні пакети SaaS) значно підвищують продуктивність команд, дозволяючи в реальному часі обмінюватися інформацією. Крім того, глобальна інфраструктура провайдерів забезпечує близькість сервісів до кінцевих користувачів: компанія може розгорнути свої системи у дата-центрах, розташованих в потрібних регіонах, що зменшує затримки і покращує досвід користувачів.

Хоча існує стереотип, що розміщення даних у зовнішнього провайдера є менш безпечним, провідні хмарні компанії пропонують глибокий і всебічний набір засобів захисту. Великі провайдери (Amazon, Microsoft, Google) інвестують значні ресурси у кібербезпеку, залучають кращих фахівців, регулярно оновлюють програмне забезпечення та оперативно встановлюють патчі безпеки. Хмарні платформи забезпечують централізоване управління безпекою, шифрування даних, системи виявлення вторгнень, резервне копіювання тощо на рівні, який часто важко досягти окремій компанії власними силами. Як результат, для багатьох організацій перехід у хмару може фактично підвищити рівень безпеки їхніх IT-систем.

Більшість хмарних провайдерів пропонують вбудовані функції резервного копіювання, аварійного відновлення та географічного дублювання даних. Зберігання інформації у хмарі означає, що у разі збою обладнання, стихійного лиха чи інших надзвичайних ситуацій дані не будуть втрачені — вони дублюються у декількох дата-центрах. Завдяки цьому компанії можуть забезпечити безперервність бізнес-процесів та швидко відновити роботу систем, що є суттєвою перевагою над традиційною локальною

інфраструктурою, де відмова серверу нерідко призводить до тривалого простою.

З іншого боку, слід враховувати й потенційні недоліки та ризики використання хмарних обчислень:

Залежність від інтернет-з’єднання та можливі простої. Для доступу до хмарних сервісів обов’язковою умовою є надійне мережеве з’єднання. Якщо інтернет-зв’язок користувача нестабільний або повільний, це безпосередньо впливає на якість роботи з хмарою. У разі відсутності з’єднання користувач взагалі не зможе дістатися до своїх даних чи застосунків. Навіть найбільші провайдери не застраховані від аварій: час від часу трапляються збої або тимчасове відключення сервісів через технічні проблеми чи стихійні лиха, що може призвести до недоступності сервісів для клієнтів, поки проблему не буде усунуто. Таким чином, бізнес критично залежить від стабільності роботи Інтернету і хмарного провайдера.

Ризик прив’язки до постачальника (Vendor Lock-In). Перенесення своїх систем у хмару конкретного провайдера може призвести до складнощів у майбутньому при зміні постачальника. Кожна платформа має власні пропрієтарні сервіси, API, інструменти, і перехід, наприклад, з AWS на Azure або навпаки, може вимагати значних зусиль з переналаштування і переробки систем. Таким чином, клієнт може виявитися “прив’язаним” до одного провайдера через високі витрати або технічні складності міграції.

Обмежений контроль над інфраструктурою. Використовуючи хмарні сервіси, компанія відмовляється від значної частки контролю за обладнанням і середовищем, на якому працюють її системи. Архітектурні рішення, оновлення обладнання, управління мережами — усім цим займається провайдер, і користувачеві доводиться покладатися на нього. Це означає, що неможливо тонко налаштовувати або оптимізувати інфраструктуру під вузькоспеціалізовані потреби так, як це можна було б зробити у власному дата-центрі. Іноді стандартні конфігурації хмари можуть не повністю відповідати вимогам

(наприклад, нестандартне апаратне забезпечення для особливих обчислень), і у таких випадках хмара може бути непридатною.

Хоча провайдери і забезпечують потужні засоби захисту, факт винесення даних за межі організації турбує багато компаній. Існують ризики несанкціонованого доступу або витоку даних через уразливості на боці хмарного середовища. Також виникають питання відповідності вимогам законодавства і регуляторів щодо збереження даних: деякі організації (банки, державні установи) обмежені у тому, де фізично можуть зберігатися їхні дані, і хмара повинна надавати необхідні гарантії локалізації і захисту інформації. Перенесення критично важливих даних у хмару потребує ретельної оцінки заходів безпеки, що їх пропонує провайдер, а також розуміння зон відповідальності: хмарні моделі спільної відповідальності означають, що частину задач (наприклад, налаштування прав доступу, шифрування даних клієнта) все одно має виконувати сам користувач.

Впровадження хмарних сервісів в існуючу IT-екосистему підприємства може бути нетривіальним завданням. Компанії часто мають legacy-системи, локальні бази даних та інші додатки, які повинні взаємодіяти з новими хмарними компонентами. Інтеграція локальної інфраструктури з хмарною (або інтеграція між хмарами різних постачальників) може викликати технічні проблеми, вимагати спеціалізованих рішень і додаткових витрат. Наприклад, забезпечення безпечного та швидкого з’єднання між локальним дата-центром і хмарою потребує налаштування VPN або виділених каналів, а узгодження форматів даних між різними системами — розробки адаптерів чи middleware-рішень.

Модель оплати за споживання, з одного боку, є перевагою, але з іншого — може обернутися проблемою, якщо нею неправильно керувати. Існує ризик, що некоректно спроєктовані або не оптимізовані хмарні системи почнуть генерувати значно більші рахунки, ніж очікувалося. Наприклад, невимкнені вчасно віртуальні машини, надмірно часті операції з диском чи мережею, або

масштабування, що вийшло з-під контролю, — все це здатне призвести до “сюрпризів” у рахунках за хмару. Тому компаніям потрібен належний моніторинг і управління витратами в хмарі. Без відповідного контролю гнучкість хмари може обернутися непередбаченими витратами, що частково нівелює економічний ефект від її використання.

Незважаючи на зазначені недоліки, світовий тренд однозначно демонструє переважання переваг хмарних обчислень над їхніми ризиками. Більшість організацій сьогодні вже не ставлять питання, чи варто переходити в хмару — натомість вони вирішують, що саме перенести та як оптимально використати хмарні можливості у своїй діяльності. Ретельне планування, вибір надійного постачальника та розуміння моделей відповідальності допомагають мінімізувати вказані ризики і успішно отримувати вигоду від хмарних технологій.

1.3 Огляд основних хмарних провайдерів (AWS, Google Cloud Platform, Microsoft Azure)

На світовому ринку хмарних обчислень наразі є багато гравців як вузько спеціалізованих на конкретних сервісах так і тих, хто надає широкий спектр інфраструктури, платформи та програмного забезпечення як послугу (IaaS, PaaS, SaaS) для бізнесу та розробників. Найбільшими на ринку хмарних провайдерів є три компанії: Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP). Кожна з них пропонує широкий спектр сервісів по всьому світу, дотримуючись схожих принципів, але маючи власні особливості як у технологічному, так і в комерційному аспектах. Далі ми коротко розглянемо кожного з цих провайдерів, приділивши особливу увагу Amazon як піонеру і лідеру ринку (рис 1.2).

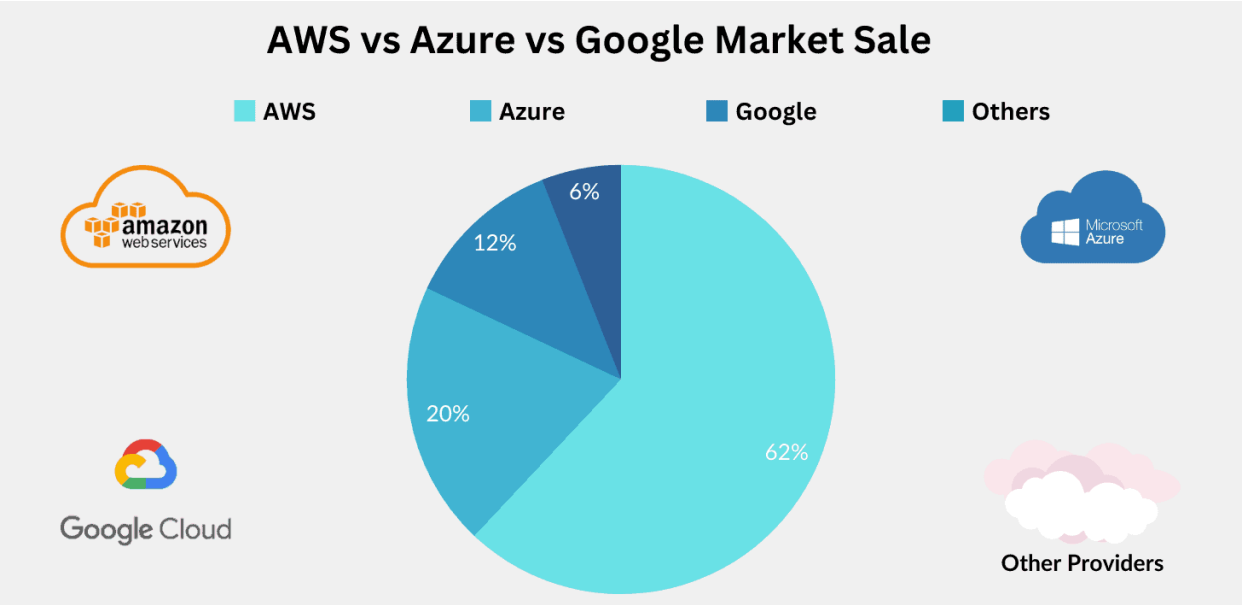

Рисунок 1.2 — Частки ринку трьох найбільших проваайдерів

Amazon Web Services (AWS) є першим великим провайдером хмарних сервісів компанії Amazon, що вийшов на ринок у 2006 році та, по суті, сформував сучасне розуміння хмарних обчислень, запровадивши модель оплати за споживання та показавши успішні випадки масштабування IT-сервісів на стороні хмарного провайдера. Наразі, Amazon Web Services — є найбільшим у сфері хмарних обчислень із приблизно третиною світового ринку. Він надає широкий набір сервісів — від базових обчислювальних потужностей (віртуальних машин), зберігання даних і мережевої інфраструктури до передових інструментів ШІ, аналітики, Інтернету речей (IoT) тощо. Інфраструктура AWS охоплює десятки дата-центрів по всьому світу: компанія має 33 географічних регіони та понад 100 зон доступності (Availability Zones) в цих регіонах, які обслуговують клієнтів у більш ніж 240 країнах і територіях. Це дозволяє розміщувати інфраструктуру сервісів ближче до кінцевого користувача та забезпечувати високу відмовостійкість шляхом географічного резервування.

AWS швидко здобув популярність та залишається лідером ринку, запропонувавши хмарні послуги раніше за конкурентів. Amazon постійно розвиває свою платформу, відому у всьому світі надійністю та технологічною

досконалістю, додаючи нові функції і сервіси, урізноманітнюючи пропозиції та можливості для клієнтів. Користувачі сервісів можуть знайти в екосистемі AWS рішення практично для будь-яких потреб (обчислення, зберігання, інструменти для баз даних, машинне навчання, DevOps-інструменти, тощо) та легко інтегрувати їх між собою.

Також суттєвою перевагою є цінова політика. Платформа залишається достатньо конкурентною, оскільки, компанія регулярно знижувала ціни на свої послуги протягом перших років розвитку та пропонує гнучкі моделі тарифікації, що приваблює як стартапи, так і великі підприємства.

AWS славиться розвиненою системою підтримки та спільноти. Мільйони активних клієнтів та величезна мережа партнерів і консультантів по всьому світу забезпечують доступність знань, документованих практик та експертної допомоги для нових користувачів.

Завдяки всім цим перевагам Amazon вже багато років утримує статус лідера, як за обсягом ринку, так і за різноманіттям та глибиною сервісів.

Microsoft Azure - друга за величиною хмарна платформа, запущена Microsoft у 2010 році, спочатку під назвою Windows Azure, а пізніше у 2014 перейменована у Microsoft Azure. Станом на зараз вона займає приблизно 25% світового ринку хмарних інфраструктурних послуг, впевнено конкуруючи з Amazon. За кількістю доступних сервісів Azure не поступається AWS, пропонуючи сотні найменувань хмарних продуктів. Microsoft активно розбудувала свою глобальну інфраструктуру, яка присутня у більше ніж 60 регіонах світу, охоплюючи всі основні географічні зони і ринки. Особливістю стратегії платформи є сильний акцент на гібридних рішеннях та інтеграції з існуючими корпоративними середовищами. Завдяки історичній ролі Microsoft у постачанні корпоративного програмного забезпечення, Azure тісно інтегрується з продуктами на кшталт Windows Server, SQL Server, Active Directory, Microsoft 365 та іншими системами, завдяки чому і стає природним вибором для організацій, які вже користуються стеком технологій Microsoft. Перехід у їхню

хмару відбувається більш плавно.

Microsoft також пропонує унікальну можливість так званої гібридної хмари через сервіс Azure Arc та Azure Stack. Компанії можуть запускати частину сервісів Azure у власних дата-центрах, отримуючи єдиний інструментарій управління для обох середовищ. Ще одна сильна сторона — розвинені засоби безпеки і відповідності вимогам (compliance). Microsoft має високу довіру серед корпоративних клієнтів у секторах фінансів, охорони здоров'я, уряду, оскільки забезпечує відповідність численним міжнародним стандартам і нормативам (ISO, GDPR, HIPAA тощо) та пропонує спеціалізовані "суверенні" хмари для державного сектору (наприклад, Azure Government в США). Таким чином, Azure часто обирають у випадках, коли безпека та регуляторна відповідність є критично важливими.

Google Cloud Platform (GCP) є третім номером на ринку з приблизно 11% його частки. Google дещо пізніше зайшла у сферу інфраструктурних хмарних послуг (перші сервіси з'явилися близько 2008 року), однак компанія має унікальний досвід створення власної глобальної інфраструктури для підтримки таких продуктів, як пошуковик Google, Gmail, YouTube, тощо. Це було використано при побудові хмарної платформи Google, завдяки чому вона славиться високою продуктивністю мереж і передовими технологіями розподілених обчислень.

GCP наразі доступна більш ніж у 40 регіонах світу і має понад 120 зон доступності та продовжує динамічно розширюватися. Google Cloud вирізняється своєю експертизою в галузі даних та машинного навчання. Компанія є лідером з розробки алгоритмів штучного інтелекту, тому в платформу інтегровано потужні сервіси для роботи з даними: великомасштабна аналітика (BigQuery), машинне навчання (AutoML, платформа TensorFlow), обробка потоків даних, штучний інтелект як сервіс (наприклад, готові API для розпізнавання зображень, мовлення, перекладу) тощо. Ці інструменти є привабливими для компаній, які сфокусовані на побудові аналітичних рішень,

AI/ML моделей та інноваційних продуктів на основі даних. Крім того, Google має історію підтримки відкритого програмного забезпечення. Саме інженери Google зробили вагомий внесок у розвиток технології контейнеризації (було розроблено систему Kubernetes для оркестрації контейнерів, що згодом була передана у open-source). Відповідно, GCP активно впроваджує та підтримує відкриті стандарти і сумісність, наприклад, дозволяє легко розгорнути контейнери Kubernetes (через сервіс Google Kubernetes Engine) і інтегрується з багатьма інструментами з відкритим кодом.

Хоча за кількістю сервісів і масштабом присутності хмарна платформа Google ще поступається AWS та Azure, вона демонструє високі темпи зростання і завойовує свою нішу, особливо серед компаній, яким потрібні спеціалізовані рішення для аналітики або які віддають перевагу екосистемі Google.

Якщо порівнювати трьох провайдерів в цілому, можна відзначити, що всі вони надають схожий базовий набір послуг (віртуальні машини, об’єктне сховище, serverless-функції, СУБД, інструменти розробки тощо) і впроваджують інновації паралельно один одному. Amazon Web Services традиційно був першим, хто пропонував нові сервіси, але конкуренти швидко скорочують розрив, і нині майже для кожного популярного сервісу AWS існує аналог в Azure та GCP. З погляду технічних особливостей, AWS виділяється найбільшою різноманітністю сервісів і довгим досвідом оптимізації своєї платформи під величезні навантаження. Azure сильний в інтеграції з корпоративним ПЗ та унікальних можливостях гібридного розгортання. GCP приваблює продуктивною інфраструктурою та спеціальними сервісами для аналітики й ШІ. Кожен з провайдерів має розгалужену партнерську екосистему і набір інструментів для міграції, моніторингу, управління витратами. Ці аспекти постійно вдосконалюються, оскільки боротьба за клієнта дуже інтенсивна у сучасному світі.

З економічної точки зору, моделі тарифікації в AWS, Azure і GCP багато

в чому аналогічні. Усі три платформи дотримуються принципу оплати за фактичне споживання (on-demand), що означає погодинну або похвилинну (в деяких випадках посекундну) оплату ресурсів без фіксованих авансових платежів. Крім того, усі провайдери мають спеціальні програми для оптимізації витрат при довгостроковому використанні ресурсів. Зокрема, AWS пропонує механізми зарезервованих інстансів та плани економії (Savings Plans), які дозволяють знизити вартість до 75% за умови зобов'язання використовувати ресурси протягом 1 або 3 років. Azure має подібні Reserved Instances для віртуальних машин і дає змогу заощаджувати при використанні власних ліцензій (Azure Hybrid Benefit). GCP запровадила модель Committed Use Discounts — знижки за фіксоване замовлення ресурсів на певний період, а також Sustained Use Discounts, які автоматично застосовуються при тривалому використанні ресурсів без зобов'язань. Окрім цього, всі згадані платформи надають можливість використовувати спотові інстанси, коли невикористані ресурси з великою знижкою, доступні без гарантії безперервної роботи (зокрема, Spot Instances в AWS, Spot VM в Azure, Preemptible VM в Google). Така гнучкість цінових моделей дозволяє підприємствам обрати оптимальний баланс між вартістю і надійністю для різних робочих навантажень.

Для залучення нових користувачів провайдери пропонують безкоштовні режими використання. AWS надає новим клієнтам Free Tier — безкоштовний рівень використання окремих популярних сервісів (наприклад, 750 годин на місяць роботи невеликої VM, певний обсяг зберігання S3, бази даних RDS, тощо) протягом перших 12 місяців, а також ряд послуг з постійно безкоштовними обмеженими обсягами (наприклад, щомісячний ліміт на виклики функцій Lambda, невеликий обсяг в DynamoDB).

Google Cloud пропонує всім новим клієнтам кредит у розмірі $300 на використання будь-яких сервісів протягом 90 днів, а також має постійно діючий безкоштовний пакет для деяких продуктів (наприклад, обмежена кількість транзакцій в Cloud Functions або певний обсяг даних у BigQuery щомісяця).

Microsoft Azure також надає бонус при реєстрації — кредит у розмірі $200 на перші 30 днів користування, плюс безкоштовний доступ протягом 12 місяців до ряду найпопулярніших сервісів (віртуальні машини, сховище BLOB, бази даних тощо у визначених межах), а також деякі послуги з безстроково безкоштовним мінімальним обсягом. Ці безкоштовні рівні дозволяють клієнтам ознайомитися з платформою і протестувати свої рішення без фінансових ризиків, після чого вже приймати рішення щодо комерційного використання.

Усі три провідні хмарні провайдери змагаються у задоволенні одних і тих самих потреб клієнтів. AWS утримує лідерство завдяки найширшому спектру сервісів, перевіреній часом надійності та великій спільноті користувачів. Azure впевнено посідає другу позицію, користуючись довірою корпоративного сегменту та пропонуючи безшовну інтеграцію з традиційними IT-середовищами. Google Cloud хоч і третій за часткою, проте вирізняється інноваціями та сильними сторонами в аналітиці даних і машинному навчанні, що робить його привабливим для певних категорій проектів. Вибір конкретної платформи часто залежить від потреб проекту: одні компанії віддають перевагу AWS за його зрілість і різноманітність, інші обирають Azure через сумісність з існуючими Microsoft-рішеннями, а хтось звертається до GCP заради спеціалізованих можливостей роботи з даними. У будь-якому разі, конкурентна боротьба між цими гігантами стимулює швидкий розвиток хмарних технологій в цілому, з чого зрештою виграють і розробники, і бізнес-користувачі.

2 ПРОЄКТУВАННЯ ХМАРНОЇ ІНФРАСТРУКТУРИ

2.1 Вибір хмарного провайдера

Amazon Web Services (AWS) — одна з наймасштабніших, якщо не наймасштабніша хмарна платформа у світі. Історія AWS розпосата у 2006 році, коли корпорація Amazon анонсувала сервіси Amazon S3 для зберігання даних і Amazon EC2 для оренди обчислювальних потужностей. Впродовж майже двадцятилітньої історії, AWS зажди додає нові сервіси, від базових безкоштовних, до машинного навчання, аналітики і власних мікропроцесорів. Такий об'єм різнобічних інструментів дає перевагу Amazon Web Services та допомагає утримувати лідируючі позиції у сфері хмарної інфраструктури. За даними Synergy Research, AWS займав майже третину, точніше 32%, на світовому ринку Iaas/PaaS у другому кварталі 2025 року, що на 9% більш аніж Microsoft Azure 23% та на цілих 20% більше за Google Cloud із їхніми 12%. Незалежні аналітики щорічно відносять сервіс від Амазону до категорії лідерів, з посиланням на найбільший вибір утиліт і найкраще інноваційність платформи. Клієнти підкреслюють, окрім великої вибірки інструментів, їхню глибоку функціональність та рівень налаштувань.

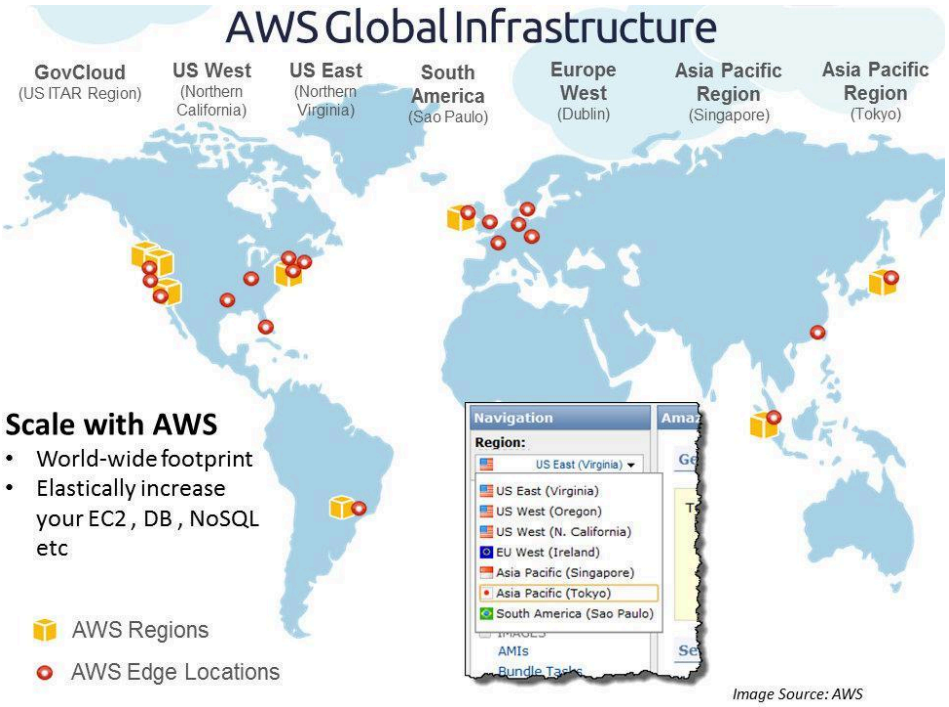

Image Source: AWS

Рисунок 2.1 — Іллюстрація зон регіонів AWS

Amazon Web Services маю глобальну, розгалужену глобальну інфраструктуру. На кінець 2025 року в експуатації знаходилось 26 географічних регіонів та понад 110 зон доступності по всьому світу. Регіони розбиті на незалежні зони доступу, у кожному щонайменше 3 зони, що забезпечує розміщення копій систем у різних дата-центрах, мінімалізовуючи ризик непрацездатності сервісу. А для прискорення доставки контенту використовується CDN-платформа Amazon CloudFront: більше 700 точок присутності (PoP) поєднані з регіональними кешуючими вузлами, що забезпечує низькі затримки при роздачі даних по всьому світу. Також, AWS активно розгортає Local Zones і Wavelength Zones — спеціалізовані зони для додатків з ультранизькою затримкою. На сьогоднішній день таких налічується вже понад 31. Користуючись такою інфраструктурою, юзери можуть розміщувати обчислювати обчислювальні ресурси якомога ближче до своїх кінцевих клієнтів, що знижує мережеві затримки. Корпорація не забуває про безпеку даних, тому всі сервіси й дата-центри пов’язані захищеною мережею AWS, а внутрішній трафік автоматично шифрується на фізичному рівні перед передачею за межі серверу. Так забезпечується надійнісь і безперервність сервісів: навіть якщо в одній зоні трапляється збій, навантаження автоматично перенаправляється на іншу зону того ж регіону.

Архітектура Amazon Web Services спроектована для еластичного масштабування та високої доступності. Застосунки розподілені на багато вузлів в різних зонах доступності, а їх навантаження балансується служюою Elastic Load Balancing. А для зручнішого масштабування є сервіс Auto Scaling, він дозволяє забезпечити потрвбну пропускну зданість у піковий період і мінімізувати витрати під час низької активності. Таким чином, при різкому зростанні трафіку, додаток додає нові віртуальні машини, а коли загрузка спадає — видаляє невикористовувані, що оптимізує відношення продуктивності до вартості. Також, важливим фактор є підтримка кількох зон доступності, яка

дозволяє уникнути єдиної точки відмови та гарантує, безперервний доступ до обчислювальних ресурсів.

Для оптимізації затримок користувачів, є глобальні сервіси маршрутизації, як Amazon CloudFront з великої кількістю PoP доставляє статичний і динамічний контнт максимально близько до споживача, Amazon Route і Global Accelerator дозволяють спрямовувати клієнтів до найменш завантаженого регіону з найнижчою затримкою.

Амазон не забули оптимізувати оплату, впровадивши варіативну систему оплати “pay-as-you-go”. Завдяки ній, користувачі опласують лише фактичне використання ресурсів, без необхідності контрактів та передплат. Базовий тариф — On-Demand в якому інстанси EC2 й інші сервіси оплачується погодинно за фіксованою сумою. Для економії витрат, AWS надають знижкові програмі такі як: Savings Plan та Reserved Instances, що передбачають фіксоване зобов’язання щодо використання за тривалий період і дозволяють заощаджувати (у середньому до 50-70% порівняно з цінами On-Demand). Крім цього, Амазон пропонує Spot-інстанси — ті самі надлишкові ресурси доступні зі знижкою до 90% від звичайної ціни. Для безкоштовного виостання є Free Tier набори із понад 100 сервісів доступними у тестовому режимі. Частина пропозицій діє 12 місяців для нових клієнтів, а частина мають необмежений безкоштовний доступ.

Величезну уваги Амазон приділили безпеці свого сервісу. Дата-центри компанії оснащені найкращим захистом, починаючи від фізичної, закінчуючи контролб доступу та цілодобовий моніторинг та архітектурою з апаратним шифруванням трафіку. Клієнти AWS отримують гнучкі інструменти контролю доступу Identity and Access Management, шифрування даних на льоту, ну і можливість контролю місця розташування інформації. Платформа відповідає численним міжнародним стандартам і вимогам, сертифікована за ISO 27001, ISO 27017, ISO 27018 та другими стандартами безпеки, ну і має сертифікати безпеки PVI-DSS, HIPPA, FedRAM… Вцілому, Amazon Web Services отримав

понад 140 сертифікацій і відповідностей. В той же час, дотримано модель спільної відповідальності Shared responsibility Model. Це означаж, що AWS несе відповідальність за безпеку фізичних серверів, мережевого устаткування, гіпервізорів і так далі. Клієнти несуть відповідальність за налаштування своїх ОС, додатків, даних і політики безпект середовища. Такий підхід спрощує управління безпекою що для користувача, що для сервісу.

AWS може застосовуватись як у навчанні, так і в бізнесі.Навіть, в Україні, після початку війни, уряд і навчальні заклади перейшли на інфраструктуру Amazon web Services, щоб зберігти освітні ресурси і забезпечити безперебійність навчання. Компанія EPAM забезпечила перехід 28 університетів на IT-інфраструктуру Амазон, завдяки чому, 260 000 студентів можуть безпечно складати тести навіть під час відключень електроенергії.

Одна з найбільших компаній, яка використовує AWS — Netflix, міжнародний стрімінговий сервіс із понад 100 млн активних користувачів. Netflix повністю базується на обчислювальних потужностях Амазон. Компанія використовує тисячі сервісів, які включають бази даних, системи аналітики, сервіси обробки відео і ще багато інших. Вони запускають понад 100 000 EC2 інстансів для безперебійного транслювання потоків по всій планеті.

Дані приклади демонструють , що AWS дуже гнучка система, на якій можна навчатись, створювати перші веб сервіси, не переживати за багато мільярдний бізнес, який повинен бути завжди в онлайні, навіть підтримує національні програми.

2.2 Вибір сервісів AWS

При проєктуванні хмарної інфраструктури для веб-додатку критично важливо правильно обрати комбінацію сервісів AWS, від якої залежить реалізація вимог безпеки, масштабованості, відмовостійкості, зберігання даних і управління ресурсами. Обчислювальні потужності надає Amazon EC2, що забезпечує масштабовані віртуальні сервери. Для зберігання даних

використовують Amazon S3 (об’єктне сховище) і EBS (блочне сховище), причому S3 забезпечує високий рівень довготривалого зберігання об’єктів з мінімальним адмініструванням, а EBS підтримує реплікацію всередині зони доступності для гарантії відмовостійкості. Організація мережевого середовища здійснюється через Amazon VPC, який надає ізольовану віртуальну мережу з повним контролем розміщення ресурсів, з’єднань та політик безпеки. DNS-маршрутизація та глобальна доступність до застосунку забезпечуються Amazon Route 53, що розподіляє трафік через георозподілені DNS-сервери з автоматичним масштабуванням для підтримки високої доступності. Усі названі сервіси взаємодіють між собою: EC2-екземпляри запускаються у VPC з налаштованими групами безпеки, доступ до них контролює AWS IAM, що керує ідентифікацією і дозволами у сервісах AWS.

Amazon EC2 надає можливість гнучкого масштабування обчислювальної потужності та налаштовується через Auto Scaling і Elastic Load Balancer для підтримки інтенсивних навантажень. Групи автоматичного масштабування EC2-екземплярів дозволяють автоматично додавати або видаляти інстанси залежно від навантаження, що підвищує доступність системи. При цьому балансувальник навантаження розподіляє запити між кількома серверами, забезпечуючи високу доступність і безпечний прийом трафіку. До кожного EC2-інстансу може бути прикріплений EBS-том — блочне сховище, яке автоматично реплікується всередині однієї зони доступності, забезпечуючи захист від відмов компонентів і підтримку стійкого зберігання даних.

Для зберігання статичного контенту (зображень, медіафайлів, файлів ресурсів) та резервних копій даних обирають Amazon S3. S3 пропонує об’єктне сховище з дуже високою довговічністю і можливістю необмеженого масштабування. Використовуючи S3, можна відмовитися від управління власною файловою інфраструктурою та скористатися захищеним сервісом з автоматичним версіюванням та шифруванням даних. У випадку реляційної бази даних доцільно обирати Amazon RDS. RDS — це керована реляційна БД, яка

забезпечує високу доступність та надійність (підтримує Multi-AZ розгортання) і може масштабуватися за допомогою реплік і обчислювальної потужності, що відповідає критеріям масштабованості та безпеки.

При проектуванні мережної та безпекової архітектури відводиться ключова роль VPC і IAM. Amazon VPC дозволяє визначити повністю ізольовану віртуальну мережу з власним IP-простором, підмережами, маршрутами та групами безпеки. Як зазначено у документації AWS, VPC «надає повний контроль над віртуальним мережевим середовищем, включаючи розміщення ресурсів, з'єднання та безпеку». За допомогою VPC можна розділити ресурси на публічні і приватні підмережі, налаштувати правила між ними та організувати безпечний вихід в інтернет через NAT. AWS IAM додає шар контролю доступу: IAM-ролі призначаються інстансам і сервісам, що забезпечує принцип найменшого привілею при доступі до ресурсів. Як зазначено на сайті AWS, IAM дозволяє «безпечно управляти ідентифікацією та доступом до сервісів і ресурсів AWS».

Amazon Route 53 виконує функції системи доменних імен (DNS) у хмарі. Цей сервіс відзначається глобальним розподілом DNS-серверів, що гарантує надійну та ефективну маршрутизацію користувацьких запитів. AWS описує Route 53 як високо доступний і масштабований DNS-сервіс, що забезпечує «надійну та ефективну маршрутизацію кінцевих користувачів до вашого веб-застосунку». У Route 53 реалізовано кілька політик маршрутизації (з урахуванням затримок, географії чи перевірок стану), які дозволяють оптимізувати продуктивність і стійкість роботи додатку. Для роздачі статичного контенту часто використовують Amazon CloudFront CDN — сервіс, що кешує об'єкти в точках присутності по всьому світу. Як зазначено в документації AWS, CloudFront «зменшує навантаження на первинний сервер і затримку, обслуговуючи більше об'єктів з найближчих до користувачів точок», що прискорює доступ до контенту та підвищує доступність ресурсу.

Нарешті, для моніторингу й управління інфраструктурою застосовують Amazon CloudWatch і суміжні сервіси. CloudWatch у реальному часі збирає метрики та логи від усіх ресурсів AWS, що дозволяє оперативно реагувати на збої та налаштовувати сповіщення. Як зазначено в офіційній документації, CloudWatch «моніторить ваші AWS-ресурси і додатки в режимі реального часу» і допомагає формувати показники для управління продуктивністю та резервуванням. Завдяки цьому системні адміністратори мають централізований огляд стану системи і можуть підтримувати її надійну роботу.

Таким чином, обираючи сервіси AWS, слід враховувати їх функціональні можливості та взаємодію. Amazon EC2 зі своїми Auto Scaling і ELB утворює динамічну основу для обробки запитів, EBS забезпечує стійке блочне сховище, а S3 із CloudFront оптимізують роботу зі статичними даними. Безпека та управління доступом гарантуються AWS IAM і VPC з гнучкими правилами мережевого доступу, а служба Route 53 підтримує глобальну доступність і балансування трафіку. Така інтеграція сервісів AWS дозволяє побудувати масштабовану, керовану та захищену хмарну інфраструктуру для вебресурсу, що відповідає всім ключовим вимогам проекту

2.3 конфігурація хмарної інфраструктури

Amazon Virtual Private Cloud (VPC) надає можливість розгортати ресурси AWS у логічно ізольованій віртуальній мережі, яка нагадує традиційну мережу у власному дата-центрі. У межах VPC задається діапазон IP-адрес (CIDR-блок), у якому створюють підмережі та налаштовують таблиці маршрутизації. Це забезпечує детальний контроль над потоками даних між підмережами і зовнішніми сервісами. Користувач отримує повний контроль над топологією хмарної мережі, що дозволяє відтворювати знайомі архітектури (наприклад, призначати IP-адреси, Інтернет-шлюзи тощо) відповідно до потреб вебресурсу, зберігаючи при цьому гнучкість та масштабованість AWS.

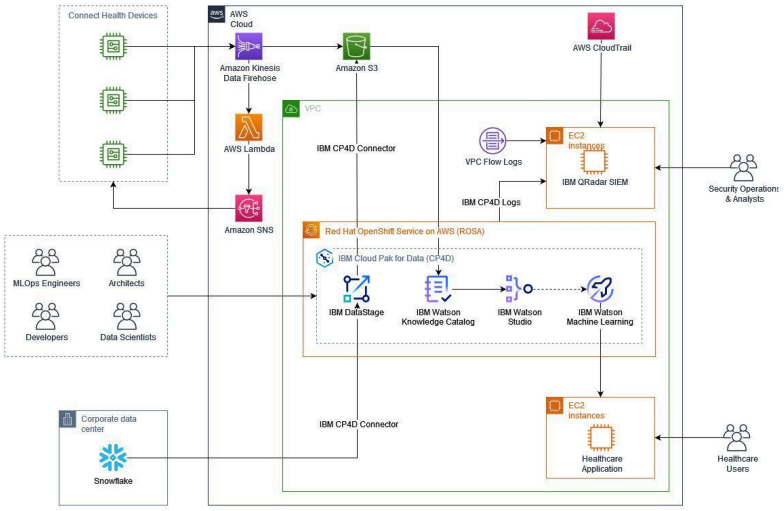

Рисунок 2.3 — Основна архітектура Amazon VPC

При створенні VPC вказують IPv4-діапазон адрес, який визначає мережевий простір VPC. Всередині цього простору створюють підмережі (subnets): кожна підмережа — це безперервний набір IP-адрес, що належить до однієї зони доступності (Availability Zone). Для підвищення відмовостійкості рекомендується формувати підмережі щонайменше в двох різних AZ. Підмережі класифікують як публічні або приватні залежно від таблиці маршрутизації: публічна підмережа має маршрут за замовчуванням до Інтернет-шлюзу (Internet Gateway), що забезпечує вихід в Інтернет, а приватна підмережа такого маршруту не має і потребує NAT-шлюзу для організації вихідного трафіку. Кожна підмережа асоціюється з таблицею маршрутизації, яка керує її вихідним трафіком (за замовчуванням підмережі зв’язані з головною таблицею VPC, але при необхідності цю асоціацію можна змінити на власні таблиці).

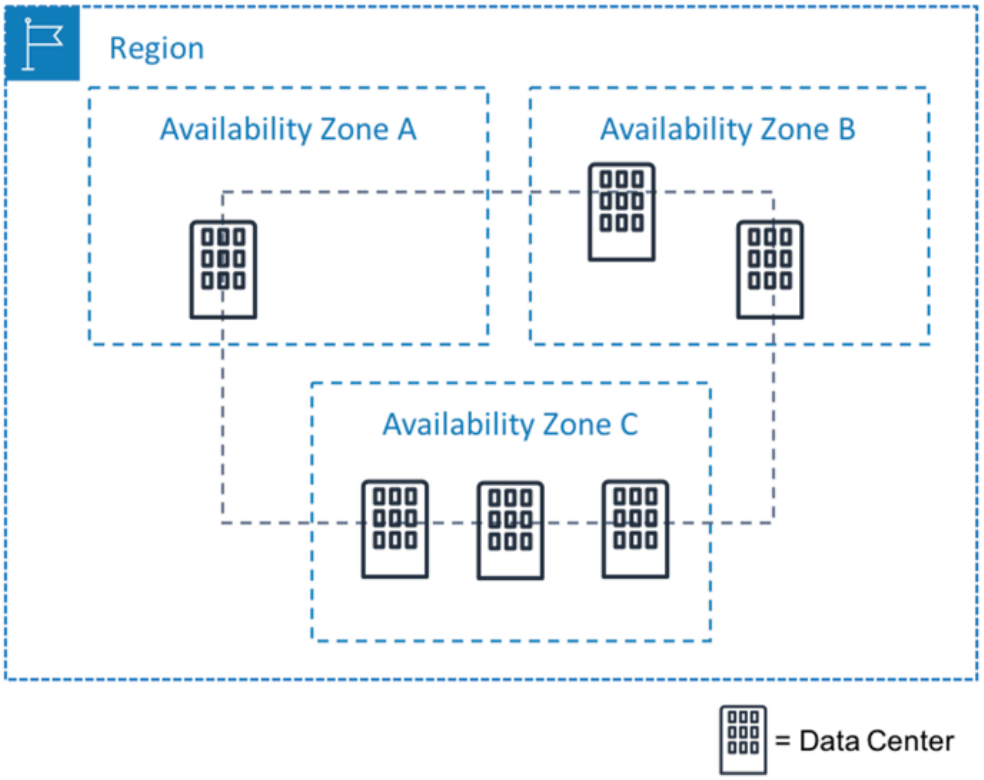

Рисунок 2.4 — Схематичне зображення Availible Zones

Після визначення мережевої структури налаштовують механізми безпеки. Групи безпеки (Security Groups) прикріплюються до кожного EC2-інстансу (або іншого ресурсу) і працюють як віртуальні фаєрволи, які контролюють вхідний та вихідний трафік за заданими правилами. При цьому групи безпеки є stateful: якщо вхідний трафік за певним правилом дозволений, то зворотні пакети автоматично допускаються без додаткових правил. На рівні підмережі використовують мережеві ACL (Network ACL): вони діють stateless і дозволяють чи блокують трафік при вході і при виході окремо. Тобто у NACL потрібно явно дозволяти обидва напрямки трафіку. Поєднання цих засобів забезпечує багаторівневий захист: групи безпеки ізолюють доступ на рівні окремих інстансів, а мережеві ACL контролюють рух пакетів між підмережами та назовні. Також у VPC передбачено налаштування брандмауерів на рівні Application (через AWS WAF) чи балансувальників (Shield) для захисту від DDoS, але базовими є SG і NACL.

Ідентифікація та управління правами доступу виконується за допомогою AWS IAM (Identity and Access Management). У цій службі створюють користувачів, групи та ролі з відповідними політиками доступу до сервісів. Для EC2-інстансів зазвичай призначають IAM-ролі, що дає змогу додаткам на серверах безпечно робити API-запити до інших сервісів AWS без зберігання статичних ключів на інстансі. Наприклад, EC2-інстансу можна присвоїти роль із правом читати дані з певного S3-бакета або писати логи у CloudWatch, вказавши це в IAM-політиці. При формуванні політик радять дотримуватися принципу найменшого привілею, обмежуючи дії лише тими API-викликами, які необхідні для роботи вебресурсу.

Після налаштування мережі створюють EC2-інстанси — віртуальні сервери для розміщення веб-застосунку. При запуску інстансу вказують AMI (образ ОС), тип інстансу, ключ SSH (для Linux) або сертифікат (для Windows) та, за потреби, IAM-роль. Інстанс асоціюється з обраною підмережею і отримує призначені IP-адреси: у публічній підмережі може мати публічну та приватну адресу, а в приватній — лише приватну. Зазвичай вебсервери розміщують у публічних підмережах (з виділеним Elastic IP чи автоматичним публічним IP), а бази даних і внутрішні служби — у приватних, без прямого доступу з Інтернету. Підключення до серверів виконується через SSH (Linux) або RDP (Windows) із використанням заздалегідь згенерованих ключів, причому групи безпеки налаштовують так, щоб відкривати лише необхідні порти (наприклад, 22 для SSH, 80/443 для HTTP/HTTPS). Для масштабування та балансування навантаження до вебшару часто додають Elastic Load Balancer (ALB/ELB), який рівномірно розподіляє запити між EC2-інстансами.

Для зберігання даних вебзастосунку використовують відповідні сервіси AWS. Для файлових систем EC2-інстансів застосовують Amazon EBS (Elastic Block Store) — довговічні блочні томи, які можна приєднувати до інстансів. EBS-томи зберігають дані незалежно від життєвого циклу віртуальних машин, до них можна підключати кілька томів до одного інстансу, а також створювати з

них снапшоти у S3 для резервного копіювання. При цьому томи підтримують шифрування за допомогою AWS KMS: якщо ввімкнути шифрування, усі дані на томі автоматично шифруються “на диску”, а передача між інстансом і томом також захищена. Для об’єктного зберігання статичних ресурсів (зображень, бекапів, відео тощо) використовують Amazon S3. S3 — це масштабоване об’єктне сховище з практично необмеженим обсягом і високою доступністю; до нього можна напряму звертатися через Інтернет по HTTPS для завантаження ресурсів. За замовчуванням усі нові об’єкти в S3 шифруються за допомогою серверних ключів Amazon (SSE-S3), а при потребі можна вмикати шифрування через AWS KMS або власні ключі (SSE-KMS/SSE-C). Крім того, AWS використовує S3 як бекенд для зберігання снапшотів EBS і образів AMI.

Для забезпечення доступності вебресурсу за доменним іменем застосовують сервіс Amazon Route 53 — глобально розподілений DNS і реєстрацію доменів. У Route 53 створюють публічну Hosted Zone та задають записи (A, CNAME тощо), які вказують домен на публічні IP-адреси або на балансувальник навантаження вашого ресурсу. Сервіс DNS Route 53 гарантує надійне та ефективне маршрутизування користувачів до вебзастосунку за рахунок глобально рознесеної інфраструктури DNS-серверів і можливостей автоматичного масштабування. Крім базової маршрутизації, Route 53 підтримує політики маршрутизації (геолокаційну, пріоритетну, з урахуванням стану кінцевих точок), що дозволяє спрямовувати трафік лише на здорові чи оптимальні цілі.

Оскільки вебресурс обробляє важливі дані, потрібно подбати про їх захист. AWS підтримує шифрування даних «на відпочинку» та «в дорозі». Наприклад, Amazon S3 автоматично шифрує всі нові об’єкти на сховищі (SSE-S3); при необхідності можна вмикати шифрування через AWS KMS або використовувати власні ключі для S3. Так само Amazon EBS дає змогу шифрувати томи на стороні сервера, що забезпечує шифрування даних на диску та захищує трафік між інстансом і томом. Для захисту даних у транзиті

використовують SSL/TLS: наприклад, при доступі до S3 або інших сервісів (через HTTPS) трафік шифрується каналом. У вебінфраструктурі зазвичай налаштовують сертифікати SSL/TLS (через AWS Certificate Manager) на балансувальнику навантаження або напряму на серверах, щоб усі HTTP-з'єднання були захищені. Окрім шифрування, важливо організувати резервне копіювання даних: це можуть бути регулярні снапшоти EBS, резервні копії баз даних, версіонування об'єктів у S3 тощо — все це дозволить відновити дані у випадку збоїв або людської помилки.

Нарешті, для підтримки стабільної роботи налаштовують моніторинг та логування. Amazon CloudWatch збирає та обробляє метрики з EC2, ELB, баз даних, S3 та інших сервісів у режимі реального часу. За замовчуванням EC2-інстанси надсилають метрики кожні 5 хвилин (можна ввімкнути деталізований моніторинг з інтервалом 1 хвилина), а всі дані зберігаються в CloudWatch до 15 місяців. За допомогою CloudWatch створюють оповіщення (alarms) на основі порогів (наприклад, завантаженість CPU або пам'яті), а також дашборди для візуалізації продуктивності системи. Для аудиту дій у акаунті AWS корисний CloudTrail, який фіксує всі API-виклики та події, допомагаючи відслідковувати зміни конфігурацій. У сумі ці інструменти забезпечують цілісну видимість стану інфраструктури, що необхідна для підтримки високої доступності і безпеки вебресурсу.

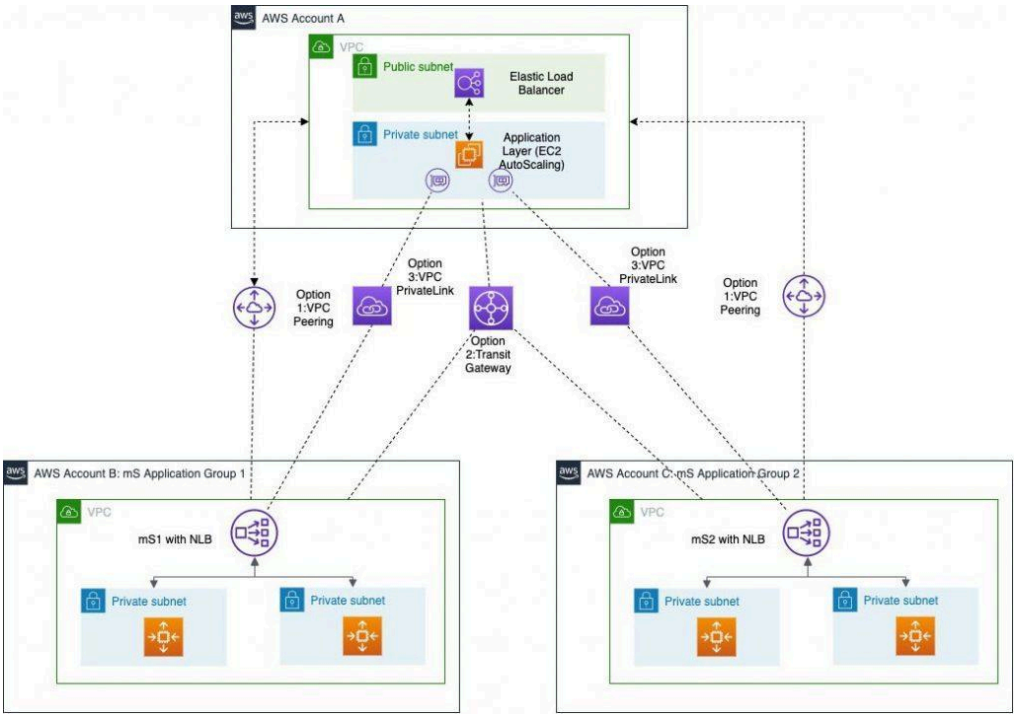

Рисунок 2.5 — Основні компоненти архітектури AWS

2.4 Створення та налаштування EC2 - інстансів

Amazon EC2 (Elastic Compute Cloud) — сервіс AWS, що надає можливість створювати віртуальні сервери (інстанси) у хмарі для хостингу вебдодатків. Кожен EC2-інстанс запускається з обраного користувачем Amazon Machine Image (AMI), який містить операційну систему та програмне забезпечення, необхідні для завантаження інстансу. При створенні інстансу користувач задає тип інстансу, який визначає апаратну конфігурацію (комбінацію процесора, оперативної пам’яті, сховищних та мережевих ресурсів) відповідно до вимог завдання. Таким чином, EC2-інстанси надають гнучкість у підборі ресурсів: для легких вебзастосунків обирають економні типи, а для ресурсоємних — потужніші інстанси.

Для зберігання даних EC2-інстанси зазвичай використовують Amazon EBS (Elastic Block Store) — мережеві блокові диски. Amazon EBS дозволяє створювати дискові томи, які підключаються до EC2-інстансів. Ці томи розміщуються у заданій зоні доступності (Availability Zone) і автоматично

реплікуються всередині неї задля захисту від апаратних збоїв. За необхідності різні типи EBS-томів (SSD або HDD різного призначення) забезпечують відповідну пропускну спроможність та I/O-показники. Крім того, EBS підтримує створення знімків (snapshots) — точкових копій даних дисків, що зберігаються в S3 і можуть використовуватися для відновлення або перенесення томів в інший регіон. Це полегшує резервне копіювання та відтворення даних у надзвичайних ситуаціях.

EC2-інстанси запускаються всередині Amazon VPC (Virtual Private Cloud) — користувацької логічно ізольованої віртуальної мережі, що подібна до мережі власного центру обробки даних. Після створення VPC вона поділяється на підмережі (subnets) — діапазони IP-адрес, що існують у конкретних зонах доступності. Інстанси, розгорнуті у цих підмережах, отримують приватні IP-адреси з відповідного діапазону. Зазвичай для доступу до Інтернету створюють публічні підмережі з підключеним Internet Gateway, а приватні підмережі забезпечують вихід через NAT-шлюз для вихідного трафіку. Зв’язок між підмережами і каналами на кшталт інтернету визначають маршрутні таблиці, які спрямовують трафік за призначенням.

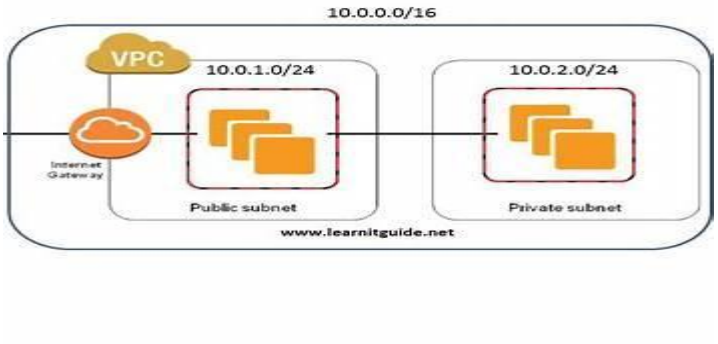

Рисунок 2. — Схема VPC з публічною і приватною підмережею

Контроль доступу на рівні мережі здійснюють групи безпеки (security groups) — віртуальні брандмауери для EC2, які визначають правила вхідного та вихідного трафіку. При створенні інстансу йому призначають одну або кілька

таких груп. Кожна група містить набір правил, які дозволяють або забороняють трафік за певними портами, IP-адресами та протоколами. Нові та змінені правила застосовуються автоматично до всіх інстансів, що асоційовані з цією групою. Завдяки цьому можна точно обмежити доступ до сервісів інстансу (наприклад, дозволити SSH-трафік тільки з певних адрес) без потреби налаштовувати фаєрвол всередині системи.

Для безпечного керування доступом до EC2-інстансів використовують ключові пари SSH та IAM-ролі. Для Linux-інстансів приватний ключ пари дозволяє безпечне SSH-підключення до машини. Публічний ключ при цьому зберігається на інстансі, а приватний — у адміністратора; втручання зловмисника вимагає контролю за приватним ключем. Для надання інстансу прав доступу до інших AWS-сервісів застосовують IAM-ролі. В ролі задають перелік дозволених дій і ресурсів, а потім асоціюють роль з інстансом. Інстанс автоматично отримує тимчасові облікові дані від AWS і може виконувати дозволені дії від свого імені (наприклад, читати дані з певного бакету S3) без необхідності зберігати статичні ключі.

EC2 підтримує кілька моделей оплати ресурсів залежно від характеру навантаження. За основною моделлю On-Demand користувач сплачує лише за фактичний час роботи інстансу (годинами чи секундами) без довгострокових зобов’язань. Це доцільно для тестових середовищ або непостійних пікових навантажень. Для стабільних і передбачуваних навантажень можна придбати зарезервовані інстанси (Reserved). Вони орендуються на один або три роки за фіксованою пониженою ставкою, що суттєво знижує вартість використання при безперервній експлуатації. Крім того, AWS пропонує спотові інстанси (Spot) — доступні за мінімальними тарифами, але їх може бути перервано при дефіциті ресурсів для On-Demand. Спотові інстанси придатні для масштабованих завдань або резервних процесів із гнучким часом виконання. Застосування комбінації On-Demand, Reserved та Spot дозволяє оптимізувати витрати залежно від потреб інфраструктури.

Для забезпечення гнучкого масштабування у відповідь на навантаження використовують Amazon EC2 Auto Scaling. За його допомогою формують Auto Scaling Groups (ASG) — логічні групи EC2-інстансів із заданими мінімальною, бажаною та максимальною кількістю екземплярів. Авто-скейлінг підтримує розмір групи в заданих межах: при збільшенні навантаження він автоматично запускає додаткові інстанси, а при зниженні — завершує непотрібні. Крім того, Auto Scaling моніторить стан «здоров'я» екземплярів (за допомогою вбудованих health checks) і автоматично замінює ті інстанси, що вийшли з ладу, щоб завжди підтримувати бажану потужність. Групи авто-масштабування також можна налаштувати з використанням різних типів інстансів (включаючи Spot) для ще більшої економії.

Моніторинг EC2-інстансів здійснюється за допомогою Amazon CloudWatch. CloudWatch збирає та обробляє сирі дані з EC2 (CPU, мережевий трафік, I/O та ін.) і перетворює їх у зрозумілі метрики у близькому до реального часу. Метрики зберігаються протягом 15 місяців, що дозволяє аналізувати історію навантажень і виявляти тренди у продуктивності. За замовчуванням дані відправляються щоп'ять хвилин, але можна ввімкнути деталізоване спостереження з оновленням щохвилини. На основі цих метрик налаштовують тривоги та автоматичні дії (наприклад, масштабування або переконфігурацію) при перевищенні заданих порогів.

Для забезпечення постійної публічної IP-адреси використовують Elastic IP. Elastic IP — це статична IPv4-адреса, що виділяється на обліковий запис AWS і залишається вашою доти, поки ви її не звільните. Вона дозволяє «прив'язати» домен напряму до інстансу незалежно від його перезапусків. У разі відмови одного інстансу ту ж адресу можна швидко переназначити на інший, що дозволяє приховати проблему від користувача і зменшити простої сервісу.

Щоб пришвидшити роботу дискових операцій рекомендується використовувати EBS-optimized інстанси. Інстанси з EBS-оптимізацією мають

спеціальну конфігурацію з виділеною смугою пропускання для операцій вводу/виводу на EBS, що мінімізує взаємні затримки між дисковими операціями та іншим навантаженням на інстанс. Це забезпечує гарантовану продуктивність дисків — наприклад, томи SSD на таких інстансах видають близько 90 % запланованих IOPS протягом 99 % часу. Багато сучасних типів EC2-інстансів (на базі Nitro) підтримують EBS-оптимізацію за замовчуванням, для решти її можна ввімкнути при запуску.

Для захисту даних EBS реалізує механізми шифрування. Amazon EBS Encryption автоматично шифрує всі томи та знімки томів за допомогою ключів AWS KMS, звільняючи розробника від необхідності власного управління ключами. Операції шифрування відбуваються безпосередньо на серверах EC2, тому дані залишаються захищеними і під час зберігання, і під час передачі між інстансом та диском. Крім того, регулярні знімки (snapshots) EBS-дисків дають змогу відновити систему або перенести її в інший регіон у разі збою. Знімки зберігаються в S3 з високою надійністю і можуть бути використані для аварійного відновлення даних.

EC2-інстанси тісно інтегруються з іншими сервісами AWS. Зокрема, для тривалого зберігання файлів та резервних копій використовується Amazon S3 — об’єктне сховище з безмежною масштабованістю та високою доступністю. Інстанси можуть напряму звертатися до даних у S3 через API або CLI, якщо їм надані відповідні IAM-права. Наприклад, вебзастосунок на EC2 може зберігати великі статичні ресурси (зображення, відео, лог-файли) у S3, щоб економити місце на локальних дисках EC2. Для рівномірного розподілу навантаження на декілька інстансів використовують Elastic Load Balancer. ELB автоматично розподіляє вхідний трафік між декількома EC2-інстансами у різних зонах доступності, при цьому постійно перевіряючи їх стан і пропускаючи трафік лише до здорових екземплярів. Для управління DNS-напрямом вхідного трафіку служить Amazon Route 53 — глобально розподілений DNS-сервіс. Route 53 перекладає доменні імена (наприклад example.com) на IP-адреси ваших

інстансів або балансувальників навантаження і, використовуючи вбудовані health-check’и, може відправляти користувачів тільки на ті ресурси, які працюють коректно. Усі ці компоненти разом дозволяють побудувати надійну, еластичну та безпечну веб-інфраструктуру на основі Amazon EC2.

3 РОЗГОРТАННЯ ВЕБ-СЕРВІСУ

3.1 Створення та налаштування EC3-інстансу

У межах практичної частини було реалізовано розгортання вебресурсу на хмарній платформі Amazon Web Services (AWS) із використанням сервісу EC2. Основними технологіями, що були застосовані під час реалізації, стали: віртуальний сервер Amazon EC2, менеджер процесів PM2, система керування версіями Git, а також протокол SSH для віддаленого доступу.

Для початку, щоб почати працювати у середовищі AWS, потрібно перейти на вебсайт Amazon (рис. 3.1).

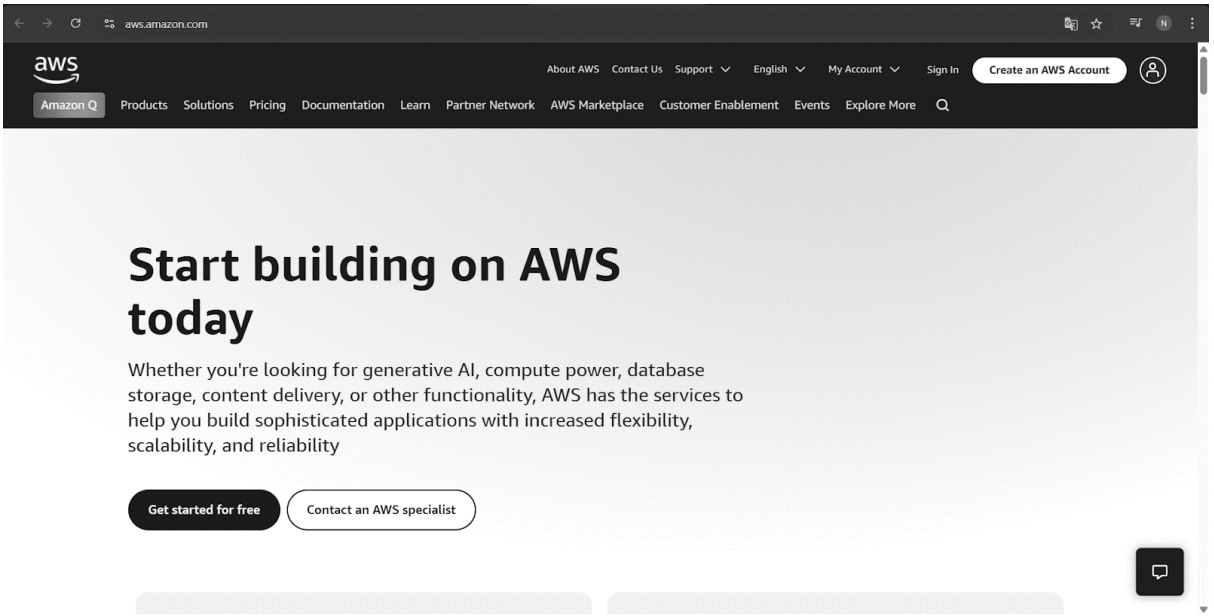

Рисунок 3.1 — Стартовий екран AWS Amazon

Далі створюєм новий аккаунт , придумавши надійний пароль . щоб ніхто не зміг отримати доступ до вашого облікового запису (рис.3.2).

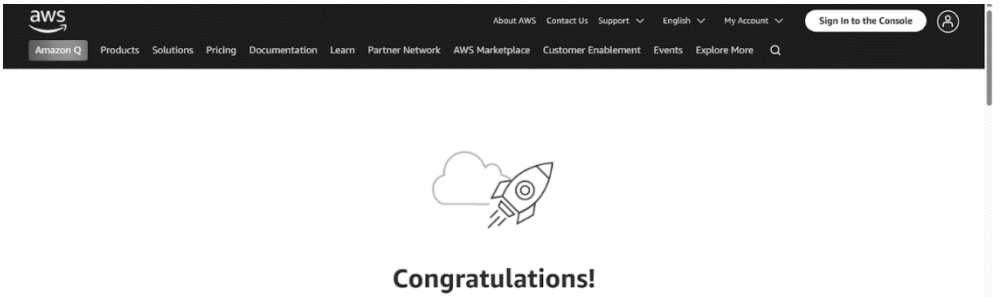

Рисунок 3.2 — Успішне створення нового акаунту

Для початку роботи потрібно перейти до AWS Manegment Console - це веб-інтерфейс, що дозоляє керувати сервісами AWS, включаючи EC2. Він забезпечує швидкий та легкий доступ до фсіх функцій, без необхідності використання командного рядка (рис. 3.3).

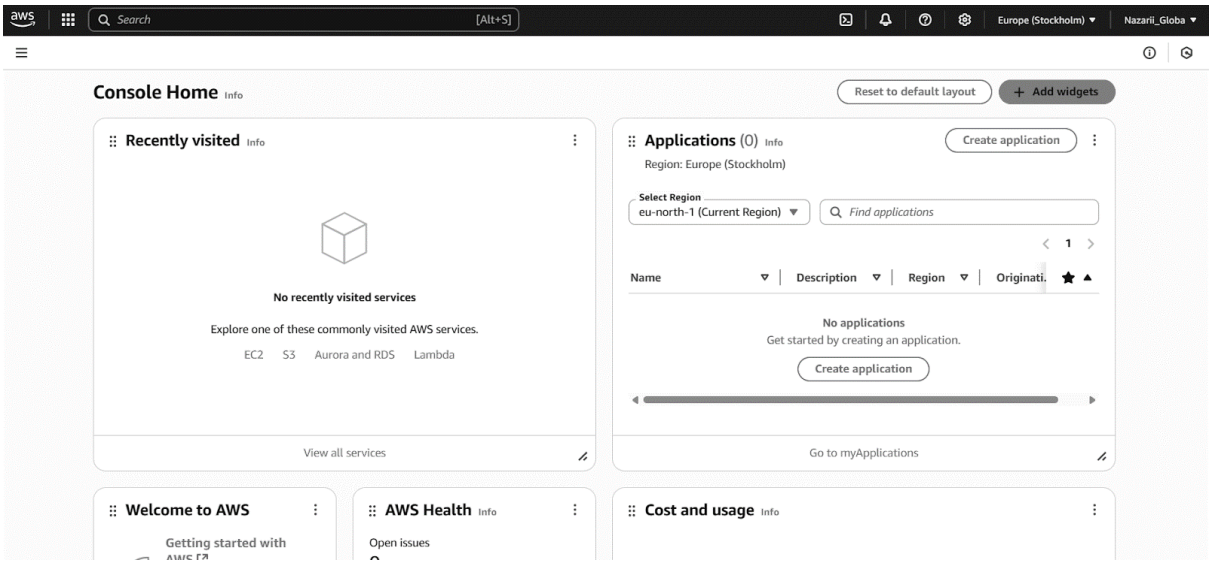

Рисунок 3.3 — Домашня сторніка консолі

Amazon EC2 (Elastic Compute Cloud) — це сервіс, що надає можливість орендувати віртуальні сервери (інстанси) в хмарі для виконання програм, вебсайтів або будь-яких інших обчислювальних завдань (рис 3.4).

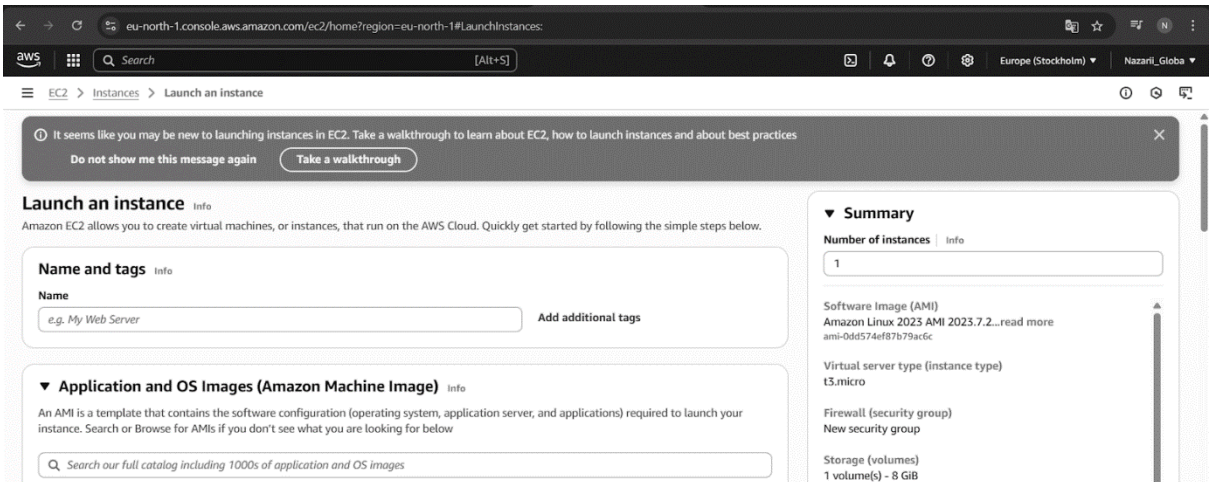

Рисунок 3.4 — Налаштування інстансу

В налаштуваннях інстансу першочергово обираєм AMI, який буде використовуватись образом.

Amazon Machine Image (AMI) — це образ, який містить операційну систему та програмне забезпечення, необхідні для налаштування і завантаження EC2-інстансу При запуску інстансу потрібно вказати певний AMI, який має відповідати обраному типу інстансу за регіоном, ОС, архітектурою процесора, типом кореневого диску і віртуалізацією.

Для бакалаврської роботи було обрано AMI Ubintu 20.04 LTS. Він зазвичай використовується для універсальних Linux-серверів. Ubuntu забезпечує 5-річну підтримку безпеки, оптимізоване ядро під AWS і публікує оновлені образи у всіх регіонах. Такий AMI підходить для веб-серверів та середовищ розробки, де потрібна стабільність і регулярні оновлення (рис. 3.5).

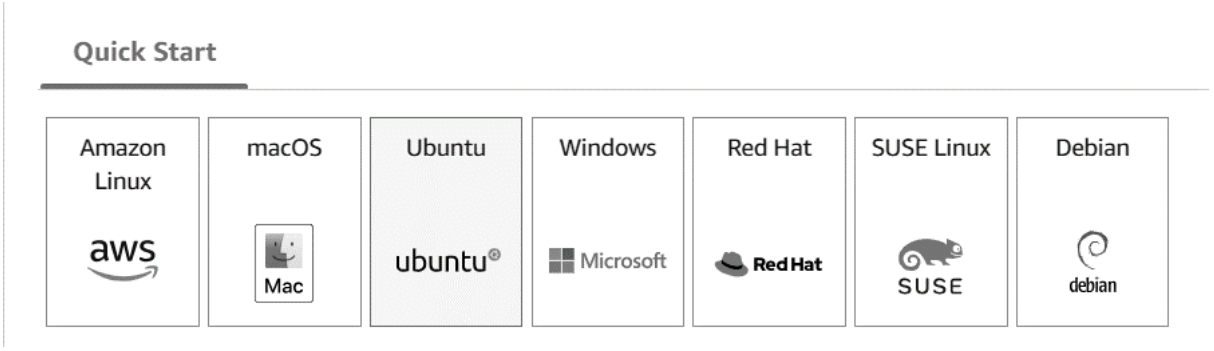

Рисунок 3.5 — Вибір AMI

Слідуючий крок — вибір типу інстансу (рис. 3.6), він визначає апаратні ресурси EC2-інстансу — поєднання процесорних ядер (vCPU), обсягу пам’яті, дискового простору та мережевої пропускної здатності. Образи згруповані у сімейства за своїми характеристиками (загального призначення — T, M; обчислювальні — C; з великим обсягом пам’яті — R; з GPU тощо). Наприклад, сімейство T3 — це недорогі загального призначення інстанси з базового рівня

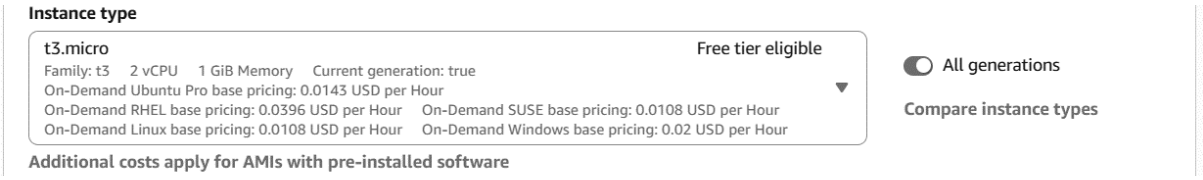

Рисунок 3.6 — Тип інстансу

Далі потрібно створити, або вибрати існуючі SSH - ключі. ВІн використовується для безпечної автентифікації користувача до віддалених серверів за допомогою протоколу SSH (Secure Shell). Цей метод є більш безпечним, ніж автентифікація за допомогою пароля, оскільки він базується на криптографії з відкритим ключем, яка унеможливлює підбір або перехоплення пароля під час з'єднання.

При створенні ключової пари AWS зберігає публічний ключ на інстансі, а ви завантажуєте приватний ключ. Приватний ключ потрібно зберігати в надійному місці (AWS його не зберігає і не відновлює, якщо його втрачено). Для Linux-інстансів підтримуються RSA-ключі з довжиною 2048 біт або ED25519, а для Windows — лише RSA

При створенні ключової пари AWS зберігає публічний ключ на інстансі, а ви завантажуєте приватний ключ. AWS не зберігає ключі та не відновлює, якщо його втрачено, тому його потрібно зберігати у надійному місці. Для Linux-інстансів підтримуються RSA-ключі з довжиною 2048 біт або ED25519, а для Windows — лише RSA, тому обираєм саме RSA.

Після цього, потрібно налаштувати віртуальний фаєрвол на рівні інстансу, який контролює вхідний і вихідний мережевий трафік. Правила Security Group дозволяють вказати протокол, порт і CIDR-диапазон для inbound (вхідного) та outbound (вихідного) трафіку. За замовчуванням при створенні Security Group немає жодних правил для вхідного трафіку (усе заблоковано), а вихідний трафік дозволений на всі адреси (0.0.0.0/0). Наприклад, щоб забезпечити доступ по SSH до інстансу, потрібно додати inbound правило TCP-порту 22 для довіреної адреси (або невеликої мережі). Щоб зробити

веб-сервер доступним, додають inbound правила для TCP-портів 80 (HTTP) і 443 (HTTPS) для адреси 0.0.0.0/0 (рис. 3.7).

На цьому, всі основні параметри для інстансу було налаштовано і залишається запустити образ (рис. Г.1).

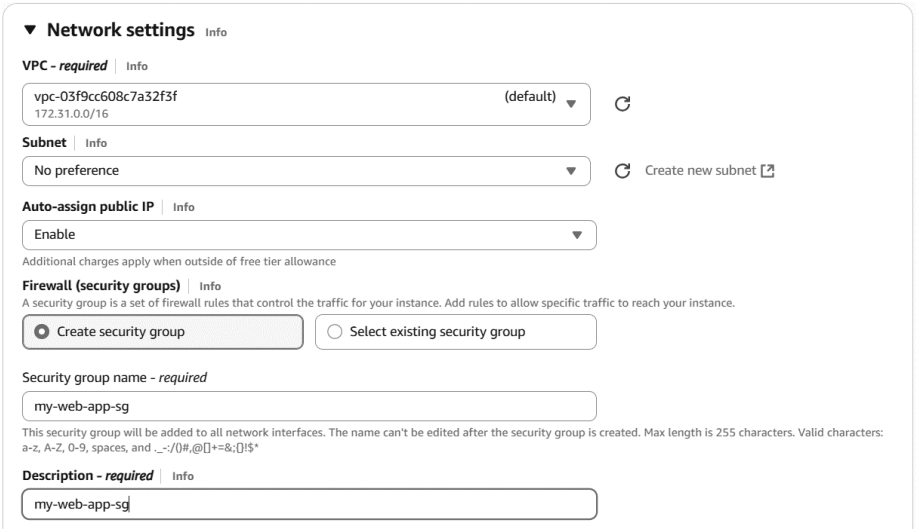

Рисунок 3.7 — Налаштування групи безпеки

3.2 Підключення до інстансу через SSH

Першим кроком розгортання є встановлення віддаленого з’єднання з EC2-інстансом за допомогою протоколу SSH. SSH (Secure Shell) забезпечує захищений шифрований канал для адміністрування віддалених Linux-серверів, тому є стандартним способом доступу до інстансу AWS. Для підключення необхідно знати публічну адресу інстансу (DNS-ім’я або IP) та ім’я користувача операційної системи на сервері (наприклад, ubuntu для Ubuntu, ec2-user для Amazon Linux тощо), а також шлях до приватного ключа SSH, який був згенерований при створенні інстансу. На локальному комп’ютері відкривається термінал і виконується команда

ssh –i “bkr-hloba-key.pem” ubuntu@ec2-ec2-16-16-68-33.eu-north-1.compute.amazonaws.com

Рисунок 3.9 — Підключення по SSH

Опція -i вказує SSH-клієнту файл приватного ключа (*.pem) для автентифікації. Частина ubuntu@ec2-ec2-16-16-68-33.eu-north-1.compute.amazonaws.com задає ім’я користувача на віддаленій машині та публічне DNS-ім’я серверу EC2, до якого здійснюється підключення.

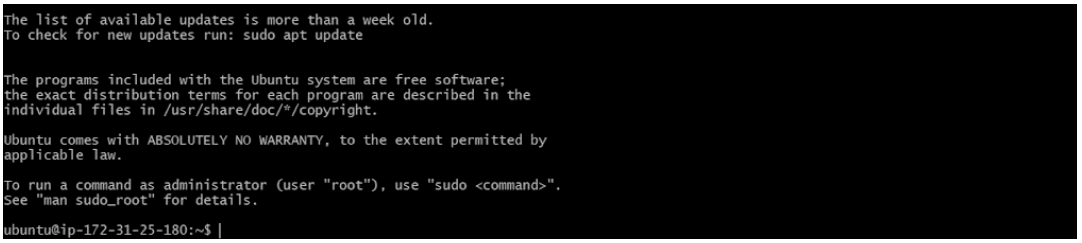

Рисунок 3.10 — Успішний результат підключення

Налаштування SSH-з’єднання є необхідним підґрунтям для подальших дій з розгортання. Отримавши доступ до серверного середовища, можна встановлювати потрібне програмне забезпечення та виконувати команди безпосередньо на віддаленому інстансі. Без SSH-доступу автоматизувати чи вручну налаштувати сервер неможливо

Перед початком роботи, обов’язково потрібно оновити систему до останньої версії та встановити усі пакети, для цього використовуєм команди

sudo apt update
sudo apt upgrade

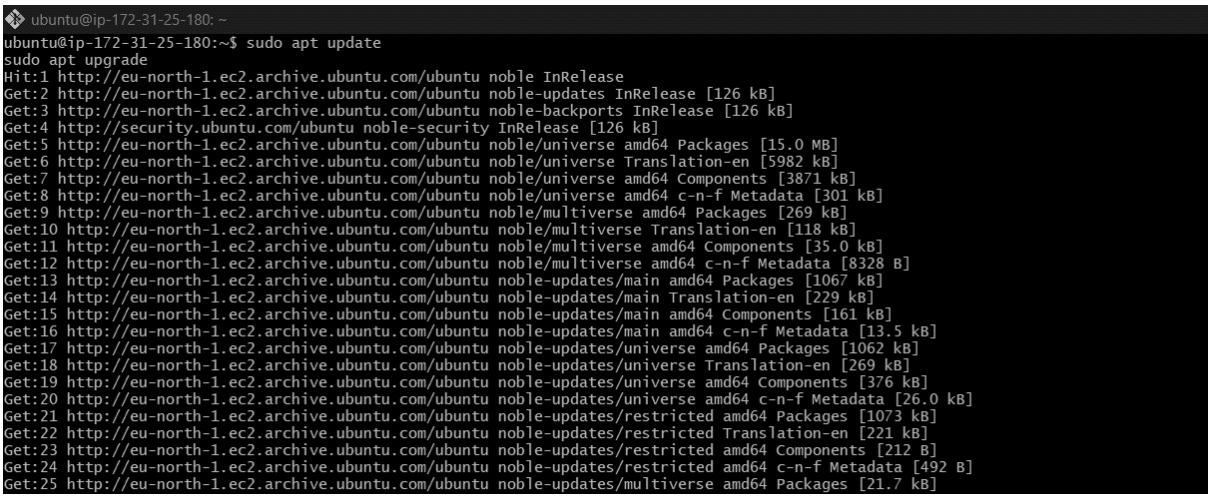

Рисунок 3.11 — Успішне оновлення системи

Після підключення до інстансу наступним кроком є встановлення Node.js — середовища виконання JavaScript, необхідного для роботи Node.js-додатку. На щойно запущеному сервері Node.js, як правило, відсутній, тому його треба інсталювати. У цій реалізації для встановлення Node.js використовується Node Version Manager (nvm) — менеджер версій Node.js, що спрощує процес інсталяції. Використання nvm доцільне, оскільки він дозволяє легко керувати кількома версіями Node.js на системі та перемикатися між ними при потребі. Це забезпечує гнучкість у виборі необхідної версії платформи Node.js і полегшує оновлення в майбутньому.

Для встановлення nvm запускається офіційний інсталяційний скрипт з репозиторію nvm на GitHub за допомогою утиліти curl

curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -

3.3 Клонування репозиторію з GIT

Ця команда завантажує з репозиторію та інсталює пакет Git разом з усіма необхідними залежностями. Після завершення встановлення доцільно перевірити версію Git (git --version), щоб переконатися, що інсталяція пройшла успішно. Наявність Git на сервері є необхідною умовою для наступного кроку — отримання коду застосунку з репозиторію.

Наступним підкроком є клонування репозиторію з вихідним кодом Node.js-додатку на EC2-інстанс. Під клонуванням мається на увазі копіювання вмісту віддаленого Git-репозиторію до файлової системи сервера. Після встановлення Git це можна зробити однією командою git clone, наприклад, якщо репозиторій розміщений на GitHub і має публічну URL-адресу.

Ця команда створить на сервері копію всіх файлів проєкту з вказаного репозиторію. В результаті виконання git clone у поточній директорії з’явиться підкаталог (як правило, з такою ж назвою, що й репозиторій), в який скопійовано весь вихідний код. Фактично, віддалений репозиторій клонується, тобто локально створюється точна копія історії та файлів проєкту. Далі слід перейти до створеного каталогу з проєктом (cd example-node-app), щоб виконувати операції саме в контексті скопійованого застосунку.

Встановлення Git та клонування репозиторію є важливими кроками, тому що забезпечують присутність вихідного коду застосунку на сервері. Без цього етапу сервер не матиме самого додатку для запуску. Використання git clone гарантує, що на EC2-інстансі буде точно та версія коду, яка підготовлена до розгортання (наприклад, остання стабільна версія з головної гілки репозиторію). Таким чином, після цього кроку у розпорядженні сервера є весь необхідний вихідний код Node-додатку.

Останнім кроком налаштування середовища є інсталяція всіх програмних залежностей проєкту та безпосередній запуск Node.js-додатку. Після клонування репозиторію у каталозі проєкту, як правило, знаходиться файл package.json, який містить перелік залежностей (бібліотек Node.js), потрібних для роботи застосунку. Ці залежності не включені до репозиторію (щоб не зберігати зайві об’ємні файли), тому їх треба завантажити з мережі на сервер за допомогою менеджера пакетів npm install.

Ця команда проаналізує файл package.json і завантажить усі пакети, зазначені у секції "dependencies" (а також "devDependencies", якщо це не виключено) із публічного реєстру npm. Менеджер npm автоматично розмістить

всі встановлені модулі у папці node_modules всередині каталогу проекту. Іншими словами, npm install встановлює *усі* залежності, перелічені в конфігурації проекту, щоб підготувати середовище для запуску застосунку. У разі успіху, після виконання цієї команди будуть доступні всі бібліотеки, від яких залежить Node.js-додаток

4 АНАЛІЗ ТА ОПТИМІЗАЦІЯ ПРОДУКТОВНОСТІ ІНФРАСТРУКТУРИ AWS

4.1 Тестування продуктивності вебдодатку на EC2

Продуктивність веб-додатку є критично важливим показником, від якого залежить задоволеність користувачів та загальна надійність системи. Навіть якщо функціональність реалізована бездоганно, надто повільна робота або збої під навантаженням можуть призвести до втрати аудиторії. Тому тестування продуктивності включається до життєвого циклу розробки, щоб переконатися, що програма здатна витримувати очікувані обсяги трафіку і своєчасно реагувати на запити. Такий підхід допомагає завчасно виявити й усунути вузькі місця, що уповільнюють роботу, забезпечуючи належну швидкодію та масштабованість ще до виходу системи в продуктивне середовище

Навантажувальне тестування, як різновид тестування продуктивності, імітує одночасну активність багатьох користувачів, дозволяючи оцінити, як веб-додаток поводиться під високим трафіком і чи не деградує його швидкодія при збільшенні навантаження

Для хмарної інфраструктури AWS, де вебдодаток розгорнуто на EC2-інстансі з використанням Node.js, такі випробування особливо важливі — вони підтверджують, що виділені хмарні ресурси і обрана архітектура здатні забезпечити прийнятний час відгуку навіть за пікових навантажень. Одним із популярних і потужних інструментів, що широко застосовуються для навантажувального тестування, є Apache JMeter

JMeter надає можливість створювати сценарії, в яких програмно генерується велика кількість паралельних HTTP-запитів до сервера, з наступним вимірюванням ключових показників його продуктивності. Важливо, що JMeter працює на рівні протоколу і не залежить від технології бекенду: з погляду цього інструмента немає різниці, на чому реалізований сервер (Node.js, PHP, Java тощо) — для JMeter це просто веб-сервер, який відповідає на HTTP-запити

В офіційній документації підкреслено, що JMeter підтримує навантажувальне тестування веб-застосунків по HTTP/HTTPS, зокрема серверів на Node.js

Таким чином, JMeter однаково добре підходить і для перевірки продуктивності REST API, реалізованого на Node.js, оскільки здатен емулювати велику кількість одночасних клієнтських з’єднань і вимірювати час відгуку та пропускну здатність API під навантаженням. Застосування цього інструменту не вимагає від інженера глибоких навичок програмування (сценарій можна налаштувати через графічний інтерфейс), що робить його зручним вибором для тестування продуктивності серверів незалежно від стеку технологій

У межах даного дослідження за допомогою JMeter проведено оцінювання продуктивності веб-додатку на Node.js, розгорнутого в AWS (екземпляр EC2). Спершу було складено тестовий план (Test Plan) — сценарій навантаження, що моделює поведінку певної кількості одночасних користувачів системи. Зокрема, у JMeter створено групу потоків (Thread Group), де задавалася кількість віртуальних користувачів (Threads) та інші параметри випробування — такі, як час нарощування навантаження (Ramp-Up) і тривалість тесту або кількість циклів запитів

Наприклад, використання Ramp-Up дозволяло вводити навантаження поступово: якщо планувалося імітувати 100 одночасних користувачів протягом 100 секунд, JMeter підключав нових користувачів приблизно по одному на секунду, уникаючи різкого стрибка навантаження на сервер

В середині Thread Group були додані семплери типу HTTP Request, які відправляли HTTP-запити до визначених кінцевих точок (endpoint) Node.js API — таким чином емулювались реальні дії клієнтів (наприклад, звернення до веб-сторінки або REST-методу). Для збору результатів у тестовий план додали Listeners — зокрема, Summary Report (або Aggregate Report), який накопичував статистику по виконаних запитах (часи відповіді, кількість успішних/неуспішних відповідей, середні величини тощо)

Кожен запуск тесту виконувався при визначеному навантаженні, після чого налаштування Thread Group змінювали — так було проведено серію випробувань при різних рівнях активних користувачів. Задіяно декілька сценаріїв: як у межах номінального очікуваного навантаження (наприклад, 50 чи 100 одночасних користувачів), так і стресові тести, що перевищують звичайний рівень (200 і більше користувачів), аби оцінити поведінку системи в граничних умовах. Для повноти картини тестувалися різні ключові функції застосунку — кілька важливих ендпоінтів API перевірялися окремо, з використанням реалістичних вхідних даних, щоб імітувати типові дії реальних користувачів

Крім того, експерименти проводилися як з короткочасним інтенсивним навантаженням (пікова перевірка), так і з тривалим поступовим збільшенням числа запитів (тест на стабільність протягом тривалішого часу). Паралельно здійснювався моніторинг стану самого сервера в хмарі (засобами AWS CloudWatch): відстежувалися завантаженість CPU, використання пам’яті та мережевий трафік під час тестів, що допомагало співставити метрики JMeter із споживанням ресурсів і виявити можливі апаратні обмеження

В результаті виконання цих навантажувальних сценаріїв JMeter зібрав ключові метрики роботи системи: зокрема, середній час відповіді (Avg), медіану (Median), мінімальний та максимальний час відповіді, стандартне відхилення, а також показник пропускної здатності (Throughput), що вимірюється у кількості запитів за секунду

Окремо фіксувався відсоток помилок (Error %), тобто частка запитів, які завершились з кодами помилок HTTP (5xx або 4xx). Ці метрики дозволяють оцінити продуктивність комплексно: Throughput відображає, скільки запитів на секунду реально обробляє сервер (тобто фактичне навантаження, яке він витримує), тоді як час відповіді показує швидкість обслуговування кожного окремого запиту

Висока пропускна здатність при малому часу відповіді свідчить про те,

що застосунок добре масштабується під навантаженням, тоді як збільшення затримки відповіді при рості числа запитів сигналізує про наближення до межі можливостей сервера. У аналізі результатів увагу приділяли не лише середньому часу відповіді, який може маскувати варіації, але й медіанному та процентильним характеристикам (90-й або 95-й процентиль), що відображають типову затримку для більшості запитів і найгірші сценарії відповідно.

Також розглядалося стандартне відхилення часу відгуку: невелике відхилення означало стабільність відповіді, тоді як великий розкид вказував на нестабільну продуктивність, коли частина запитів обслуговується значно довше за середнє значення. Залежність середнього часу відповіді Node.js-додатку від кількості одночасних користувачів (приклад результатів випробування). Як показано на графіку, при невеликій кількості паралельних запитів сервер забезпечує швидку відповідь (сотні мілісекунд). У міру зростання числа одночасних користувачів середній час відповіді певний час залишається на прийнятному рівні, однак після досягнення певного порогу навантаження починає різко збільшуватися.

З практичного погляду, різке зростання часу відповіді при подальшому збільшенні навантаження означає деградацію користувацького досвіду — запити починають виконуватися занадто повільно, і користувачі можуть стикатися з таймаутами. Пропускна здатність (кількість оброблених запитів за секунду) в залежності від кількості паралельних користувачів.

Спершу throughput зростає майже лінійно разом зі збільшенням числа користувачів — тобто, що більше одночасних сеансів, то більше запитів на секунду успішно обробляє сервер. Однак на деякому рівні навантаження крива досягає плато: додавання нових користувачів уже не призводить до зростання пропускної здатності. Ця точка відома як «точка насичення» — момент, коли система повністю завантажена і працює на межі своїх ресурсів

Після її досягнення подальше збільшення навантаження не дає виграшу в пропускній здатності, натомість різко погіршується час відповіді, і починають

фіксуватися помилки обробки запитів.

В ході тестування, наприклад, було виявлено, що при перевищенні ~N одночасних користувачів сервер починає відкидати частину запитів з помилками HTTP 5xx — ознака того, що він не встигає обробити всі звернення в рамках таймаутів. Така поведінка відповідає очікуваному сценарію: спершу система масштабувалась під навантаженням майже лінійно, потім досягла пікової продуктивності, після чого подальше навантаження призвело до її перевантаження та деградації показників. Навантажувальне тестування за допомогою JMeter дало чітке уявлення про межі продуктивності досліджуваного Node.js-додатку в середовищі AWS[19].

На основі отриманих результатів можна зробити практичні висновки щодо необхідності оптимізації та масштабування. Зокрема, було визначено приблизний поріг користувацького навантаження, до якого система працює стабільно (із прийнятними часами відповіді і без помилок).

Якщо прогнозується, що у реальній експлуатації кількість одночасних користувачів може перевищити цей поріг, постає завдання масштабування: вертикального (наприклад, перехід на більш потужний тип EC2-інстансу) або горизонтального (додавання додаткових інстансів Node.js за балансувальником навантаження). Аналіз метрик JMeter дозволив виявити найбільш вузькі місця — наприклад, якщо основним обмеженням виступає CPU, доцільно збільшити його частоту чи кількість ядер, або оптимізувати алгоритми на стороні Node.js, що споживають найбільше процесорного часу.

Якщо ж продуктивність лімітується затримками доступу до бази даних чи інших зовнішніх сервісів, результати тестів вкажуть на доцільність відповідних оптимізацій (кешування, збільшення пулу з’єднань тощо). Отже, використання JMeter для навантажувального тестування Node.js API на AWS не лише підтвердило здатність системи обробляти заданий обсяг трафіку, а й надало цінні дані для подальшого вдосконалення архітектури.

4.2 Тестування стійкості до DDoS-атак з використанням BreakingPoint

Cloud

Призначення тестування DDoS-стійкості. Розподілені атаки відмови в обслуговуванні (DDoS) становлять серйозну загрозу для веб-ресурсів: зловмисники генерують масовий трафік з багатьох джерел, щоб перевантажити інфраструктуру і зробити сервіс недоступним для легітимних користувачів. У контексті хмарних інфраструктур, які самі по собі забезпечують еластичність і базовий захист, проведення тестування на стійкість до DDoS-атак має на меті перевірити, наскільки правильно сконфігуровано додаткові засоби захисту і архітектуру додатка.

Провайдери хмар (наприклад AWS) впроваджують базові механізми захисту на рівні своєї інфраструктури (рівні L3/L4), але відповідальність за надійну роботу самого застосунку розподілена: розробник має переконатися, що увімкнено всі необхідні опції (WAF, розширений захист тощо) і що архітектура здатна витримати навантаження атаки. Отже, тестування DDoS-стійкості дозволяє виявити слабкі місця системи, перевірити ефективність захисних механізмів та готовність інцидент-відповіді до реальної атаки. На відміну від тестування продуктивності (як у розділі 4.1, де моделюється легітимне навантаження користувачів), випробування на відмовостійкість під DDoS фокусується на екстремальних, зловмисних сценаріях навантаження і перевіряє саме стійкість системи до відмов, а не лише швидкодію.

DDoS-атаки є однією з провідних причин простою хмарних сервісів у бізнесі, і частота таких атак зростає. Тому регулярне симуляційне тестування DDoS вважається кращою практикою для хмарних додатків: воно дає змогу проактивно перевірити, чи спрацюють механізми автозахисту і чи збереже система доступність під час атаки. Платформи для симуляції DDoS-атак (BreakingPoint Cloud та аналоги).

Для безпечного моделювання DDoS-навантаження використовуються спеціалізовані програмно-апаратні платформи. Сучасним прикладом є BreakingPoint Cloud — хмарний сервіс компанії Keysight (раніше Ixia), що

надається як SaaS-рішення. BreakingPoint Cloud дозволяє генерувати великомасштабний трафік атаки через розподілених агентів у хмарі, без необхідності встановлення локального ПЗ. Важливо, що платформа містить вбудовані засоби безпеки: перед запуском вона вимагає підтвердити право власності на цільову адресу (перевіряє, що IP належить клієнту у його хмарному обліковому записі), аби уникнути ненавмисного спрямування трафіку на чужі ресурси. Інтерфейс BreakingPoint Cloud спрощений для користувача та містить готові профілі атак — типові сценарії різної інтенсивності і тривалості, що мінімізує ризик помилки конфігурації при запуску тесту.

Платформа масштабується еластично залежно від обсягу потрібного трафіку: можна моделювати як відносно малі атаки, так і дуже великі (у партнерстві з провайдерами хмар, такими як Azure або AWS) для перевірки стійкості на рівні терабіт. Понад 20 років досвіду компанії Ixia у сфері тестування мереж було закладено в BreakingPoint Cloud, завдяки чому вона регулярно оновлюється новими видами атак і дозволяє отримати «реальний трафік в лабораторії» — тобто генерує як шкідливий DDoS-трафік, так і фон легітимного навантаження, щоб перевірити систему в умовах, максимально наближених до реальних. Окрім BreakingPoint Cloud, існують й інші засоби для симуляції атак. LOIC (Low Orbit Ion Cannon) — один з перших і найбільш відомих інструментів для генерування DoS/DDoS-трафіку, що став популярним завдяки відкритому вихідному коду та простоті використання.

Він дозволяє з окремого комп’ютера виконувати UDP-, TCP- чи HTTP-флудинг (тобто засипати ціль безперервними пакетами або запитами) і таким чином перевантажувати її ресурси. LOIC надає користувачу змогу налаштувати інтенсивність атаки (кількість запитів, потоків) і тип протоколу, має зрозумілий графічний інтерфейс, а також режим “hive” для координації кількох копій через IRC. Історично цей інструмент використовувався як для стрес-тестування власних сайтів, так і зловмисниками (зокрема групою Anonymous) для масових протестних атак. Проте можливості LOIC обмежені:

він генерує трафік з одного вузла і не здійснює розподіленої атаки сам по собі — для справжнього DDoS потрібна велика кількість маштабованих “зомбі”-клієнтів або ботнет.

Тому для професійного тестування стійкості його застосовують рідко; натомість на практиці використовуються більш потужні платформи або сервіси. Radware — відомий постачальник рішень кібербезпеки — також надає послуги з тестування DDoS-захисту.[20] Зокрема, у межах свого хмарного сервісу захисту Radware пропонує одноразовий сеанс валідації: команда експертів Radware (Emergency Response Team, ERT) за попередньою домовленістю проводить контрольовану імітацію атаки на ресурси клієнта, що захищені їхнім же сервісом. Такий тест дозволяє замовнику перевірити налаштування безпеки, підтвердити, що недавні зміни в мережевій інфраструктурі не послабили захист, і задокументувати ефективність захисного рішення для дотримання вимог регуляторів.

В ході тесту Radware може залучати сторонні інфраструктури для генерації трафіку (наприклад, оркеструвати ботнет-атаки в хмарі) тривалістю до кількох годин, при цьому інженери компанії моніторять систему в реальному часі. По завершенні надається детальний звіт із висновками та рекомендаціями щодо покращення захисту. Фактично, Radware поєднує симуляцію атаки з експертним аудитом: їхні фахівці аналізують спрацювання засобів захисту (наприклад, чи правильно відбувся відсів трафіку на їхніх “скраббінгових” центрах із сумарною пропускною здатністю 12 Тбіт/с) і допомагають налаштувати систему оптимально під загрози. PacketStorm — ще один інструмент, який часто згадують у контексті тестувань мереж.

Історично PacketStorm Communications розробляла апаратно-програмні емулятори мереж (WAN-емулятори) для лабораторного відтворення реальних умов трафіку. Ці рішення дозволяли генерувати та модифікувати IP-трафік, вводити затримки, втрати пакетів, змінювати пропускну здатність — тобто симулювати стресові умови в мережі. Хоча сам по собі емулятор PacketStorm не

є “DDoS-генератором” у класичному розумінні, він може використовуватися у зв’язці з іншими інструментами для точного відтворення сценаріїв атаки. Окрім того, спільнота Packet Storm Security — це онлайн-архів скриптів та утиліт для тестування безпеки, де присутні і численні засоби для DoS/DDoS (такі як скрипти HOIC, Slowloris, HULK, R.U.D.Y. тощо). Таким чином, аналітики та розробники можуть комбінувати професійні платформи (BreakingPoint, Radware) з відкритими інструментами (LOIC, скрипти з Packet Storm) для побудови повноцінної картини стійкості інфраструктури до різних видів атак. Типи DDoS-атак та їх моделювання. DDoS-атаки поділяють на декілька основних категорій за характером впливу на ціль, і повноцінне тестування стійкості повинно охоплювати всі ці типи.

Перший клас — об’ємні атаки (Volumetric), мета яких — вичерпати пропускну здатність каналу або ресурси мережевих пристроїв шляхом надмірного трафіку. Прикладом є UDP-флуд або ампліфікаційні атаки типу DNS/NTP Amplification, коли зловмисник використовуючи підроблені запити змушує потужні сервери відправляти масові відповіді жертві, створюючи лавину трафіку. Об’ємні атаки вимірюються в Гбіт/с або млн пакетів/с і спрямовані на перевантаження мережевих каналів[21].

Другий клас — атаки на рівні протоколів (Protocol Attacks), що експлуатують уразливості або особливості мережевих протоколів для виснаження ресурсів серверів, балансувальників чи брандмауерів. Класичний приклад — SYN Flood (атака на TCP-handshake), де атакувальник шле величезну кількість запитів SYN, не завершуючи встановлення з’єднання, чим змушує сервер тримати напіввідкриті сесії і витрачати пам’ять та обчислювальні ресурси. Інші приклади — Ping of Death, Smurf чи фрагментаційні атаки, які можуть спричинити збої у мережевому стеку.

Такі атаки діють на рівнях L3/L4 (мережевому та транспортному) і націлені радше на інфраструктуру, ніж на сам додаток. Третій ключовий клас — атаки на прикладному рівні (Application Layer Attacks). Вони імітують

поведінку справжніх клієнтів, але у масових масштабах, націлюючись на слабкі місця логіки додатку або ресурсномісткі операції. Наприклад, HTTP-flood — безперервний потік HTTP GET або POST запитів до веб-сервера (зокрема, до важких сторінок чи API), що виснажує потужності веб-сервера або бекенд-бази даних. Інші відомі приклади — атаки типу Slowloris (відкриття великої кількості HTTP-з'єднань і вичікування, щоб вичерпати пул підключень) чи R.U.D.Y. (повільне надсилання довгих POST-запитів). Хоч такі атаки не створюють значного обсягу трафіку, вони небезпечні тим, що прямо б'ють по рівню застосунку і можуть пройти повз мережеві фільтри. Метою є вичерпати потужність CPU/RAM веб-сервера, потоки додатку або інші обмежені ресурси, роблячи сервіс недоступним для нормальних користувачів. Для повного тестування симуляційні платформи генерують усі зазначені типи атак. BreakingPoint, наприклад, має вбудовану бібліотеку з понад 190 тисячами відомих сигнатур атак та шкідливих навантажень.

В ході одного сценарію вона може одночасно відтворювати мікси трафіку: наприклад, 80% легітимних HTTP-запитів до веб-сайту, 10% — SYN-флуд на той же сервер і 10% — спроб експлуатації відомих уразливостей або надсилання шкідливих пакетів. Така комбінація дозволяє побачити, як система поводиться під час багатовекторної атаки — коли одночасно йде навантаження і на мережу, і на протокольний рівень, і на сам застосунок.[22] Аналогічно, інші інструменти (LOIC, скрипти з Packet Storm) можуть бути використані для окремих видів навантаження — наприклад, генерування лише HTTP-флуду або лише DNS-шторму — у разі, якщо потрібно перевірити вразливість до конкретного типу атаки.

Важливо переконатися, що під час тестування покрито всі класи загроз, адже архітектура захисту від DDoS повинна мати багатоешелонний характер (мережеві екрануючі засоби для volumetric і protocol-атак, та WAF/капчі/фільтри для application-атак). Приклад сценарію атаки на AWS-інфраструктуру. Розглянемо гіпотетичний сценарій тестування, коли ціль — веб-застосунок,

розгорнутий на AWS. Припустимо, застосунок працює на одному або кількох екземплярах Amazon EC2 за балансувальником навантаження (Elastic Load Balancer, ELB). Для моделювання атаки можна використати BreakingPoint Cloud або подібний сервіс, який запустить трафік з хмари інтернет-провайдерів прямо на публічну точку входу додатку (наприклад, на DNS-ім'я балансувальника або IP-адресу EC2-інстансу). Сценарій 1: імітація масованої атаки на рівні TCP.

Наприклад, генерується SYN-флуд на порт 443 (HTTPS) балансувальника із інтенсивністю ~100 000 пакетів на секунду протягом 10 хвилин. Такий потік неповних TCP-запитів має перевірити, чи балансувальник і захисні системи (AWS Shield) зможуть відфільтрувати сплеск трафіку і не допустити вичерпання ресурсів серверів. Сценарій 2: моделювання атаки на рівні застосунку — припустимо, HTTP-GET-флуд на домашню сторінку веб-сайту. BreakingPoint може розгорнути сотні віртуальних клієнтів, які одночасно надсилатимуть тисячі HTTP-запитів в секунду до ELB, імітуючи ботнет. Мета — перевірити, чи спрацюють правила AWS WAF (якщо налаштовані) для блокування надмірного трафіку, та наскільки зросте навантаження на самі EC2-інстанси (чи не вичерпаються потоки, CPU тощо). Сценарій 3: комбінована атака[23]. Наприклад, паралельно з HTTP-флудом запускається DNS Amplification на домен, прив'язаний до того ж застосунку: публічні DNS-сервери Route 53 отримують значно більше запитів, ніж звичайно, з метою перевірити стійкість DNS-шару. Усі ці сценарії можна поступово нарощувати за інтенсивністю — від малого навантаження до екстремального — та відслідковувати, за якого порогу система починає деградувати.

Важливим параметром є тривалість: реальні атаки можуть тривати години і дні, тому корисно перевірити не тільки короткочасний сплеск, а й стійкість протягом тривалого часу (наприклад, годинний безперервний флуд) — чи не виникне витік пам'яті, чи спрацюють механізми авто-масштабування. Механізми реагування та автоматичного захисту в AWS. Хмарна платформа

AWS надає ряд вбудованих засобів, які покликані автоматично пом’якшувати DDoS-атаки або сприяти стійкості сервісів. На базовому рівні всі клієнти AWS отримують захист AWS Shield Standard без додаткової оплати: це сукупність технологій, інтегрованих у мережу AWS, що автоматично виявляють та пом’якшують найпоширеніші інфраструктурні атаки (рівнів 3 і 4) — такі як SYN-флуд, UDP-флуд, відбивні атаки через публічні служби тощо.

Shield Standard працює на рівні магістральної мережі AWS і покликаний захистити саму хмарну інфраструктуру від перегрузок; проте для тоншого налаштування захисту конкретного додатку AWS пропонує розширений рівень — AWS Shield Advanced. Shield Advanced — це платний сервіс, що підключається для конкретних ресурсів (Elastic IP, балансувальники, CloudFront дистрибуції, тощо) і надає розширені можливості: по-перше, цілодобовий моніторинг та підтримку від команди реагування AWS SRT (Shield Response Team) у разі складної атаки. По-друге, гнучкіші пороги виявлення та швидше спрацьовування захисту — правила фільтрації трафіку підлаштовуються під нормальний профіль конкретного додатку і можуть реагувати на відхилення швидше, ніж стандартні механізми. По-третє, Shield Advanced тісно інтегрується з веб-брандмауером AWS WAF: він дозволяє автоматично застосувати WAF-рецепти для виявленої атаки. Наприклад, якщо фіксується L7-атака (HTTP-флуд), Shield Advanced може автоматично згенерувати правило WAF, яке почне блокувати шкідливі запити (за певними ознаками або по IP-адресах зловмисників)[24].

Цей механізм відомий як automatic application layer DDoS mitigation — фактично, штучний інтелект аналізує трафік та самостійно створює фільтри у WAF для відбиття атаки без ручного втручання адміністратора. Ще одна важлива перевага Shield Advanced — захист від економічних витрат: якщо під час атаки ваші ресурси масштабувались або було передано значно більше даних через CloudFront/ELB, AWS може компенсувати такі витрати (що критично, адже атака може спричинити великі рахунки). AWS також надає Web Application

Firewall (AWS WAF) — це керований брандмауер для рівня HTTP/HTTPS, де можна прописувати правила блокування запитів за різними умовами (по IP, по URI, по вмісту запиту, по географії тощо).

В контексті DDoS WAF корисний для відсікання більш «розумних» атак на прикладному рівні. Наприклад, можна створити rate-based правило, яке автоматично блокує клієнта (IP-адресу), якщо той зробив понад N HTTP-запитів за хвилину. Це ефективно зупиняє прості HTTP-флуд атаки, коли один бот починає засипати сервіс запитами. WAF також здатен розпізнавати і блокувати відомі шаблони шкідливих звернень (SQL-ін’єкції, XSS тощо), що може бути корисним, якщо DDoS супроводжується експлуатацією вразливостей.

Для захисту API є спеціальний режим (AWS WAF для API Gateway) з контролем частоти викликів. Під час тестування важливо переконатися, що WAF налаштовано правильно: наприклад, не занадто агресивно, щоб не блокувати легітимний сплеск трафіку, але й достатньо чутливо, щоб зупинити атаку.

Ще один ключовий механізм стійкості — автоматичне масштабування (Auto Scaling). AWS Auto Scaling дозволяє динамічно додавати нові EC2-інстанси в групу, коли існуючі перевантажені (за CPU, пам’яттю, мережевими метриками), і відключати зайві, коли потреба відпала. Під час DDoS-атаки авто-масштабування може тимчасово збільшити кількість ресурсів, щоб обробити підвищене навантаження, тим самим запобігши падінню сервісу. Наприклад, якщо веб-сервери під навалою великої кількості запитів, Auto Scaling розгорне додаткові екземпляри, розподіляючи трафік між більшою кількістю вузлів.

Це не зупиняє атаку як таку, але підвищує ймовірність, що легальні користувачі все ще зможуть отримати відповідь (хоч повільнішу) замість повного збою[25]. В ідеалі, авто-масштабування поєднується з фільтрацією: поки WAF/Shield відсіюють частину трафіку, нові інстанси беруть на себе залишок навантаження. Тестування може показати, наскільки швидко

спрацьовує масштабування і чи вистачає максимальних лімітів (наприклад, чи не впирається група в межу кількості інстансів).

Слід також врахувати балансувальники навантаження (ELB): вони самі по собі розроблені витримувати великий обсяг трафіку, розподіляючи його по бекендах, та мають захист від деяких протокольних атак. Однак є випадки, коли надмірна кількість нових з'єднань може перевищити спроможність ELB обробляти queue, тому AWS рекомендує використовувати Global Accelerator (рис. 4.1) або CloudFront спереду, щоб наперед відсіяти частину трафіку і використати глобальну мережу AWS для поглинання атаки.

На периметрі — глобальні служби (DNS-сервер Amazon Route 53 та мережа доставки контенту Amazon CloudFront), які працюють під захистом AWS Shield Advanced (рис. 4.1). AWS WAF застосовується на рівні CloudFront або на балансувальнику. У регіональному сегменті (VPC) розгорнуто балансувальник навантаження (Elastic Load Balancing), що розподіляє трафік на EC2-інстанси. Така багаторівнева схема (edge-сервіси + WAF + авто-масштабування) ізолює атаки ближче до джерела та зменшує навантаження на основні ресурси.

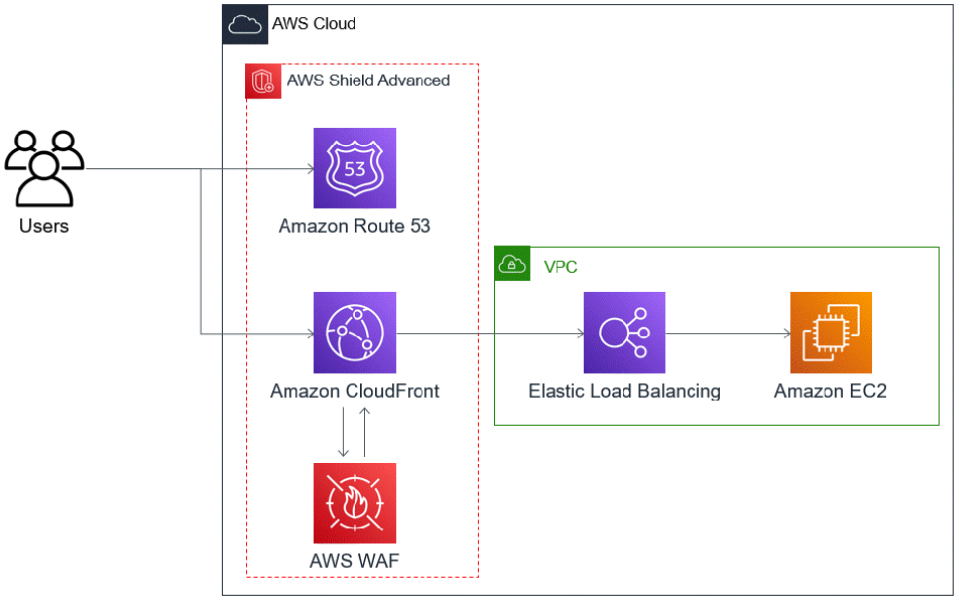

Рисунок 4.1 — Типова архітектура веб-застосунку на AWS з врахуванням DDoS-захисту.

На периметрі — глобальні служби (DNS-сервер Amazon Route 53 та мережа доставки контенту Amazon CloudFront), які працюють під захистом AWS Shield Advanced. AWS WAF застосовується на рівні CloudFront або на балансувальнику. У регіональному сегменті (VPC) розгорнуто балансувальник навантаження (Elastic Load Balancing), що розподіляє трафік на EC2-інстанси. Така багаторівнева схема (edge-сервіси + WAF + авто-масштабування) ізолює атаки ближче до джерела та зменшує навантаження на основні ресурси. Під час тестування сценаріїв, подібних наведеним вище, така архітектура дозволяє оцінити, наскільки кожен шар виконує свою функцію: чи розсіює Route 53 та CloudFront трафік атаки по глобальній мережі (із сумарною пропускною здатністю в кілька Тбіт/с), чи відсіює WAF підозрілі запити (наприклад, по IP або по сигнатурах) чи справляється балансувальник з залишковим навантаженням, і чи додає Auto Scaling достатньо екземплярів EC2 для обслуговування легітимного трафіку. Таке комплексне випробування дає цінну інформацію для удосконалення архітектури. Можливо, результати вкажуть на потребу увімкнути додаткові правила WAF, або використати кешування на CloudFront, або підняти ліміти авто-масштабування тощо — все це краще виявити під час планового тесту, ніж у розпал реальної кібератаки.

Значення тестування DDoS-стійкості для бізнесу і розробників. Проведення симуляційних DDoS-атак в контрольованих умовах має велику практичну користь. По-перше, воно дозволяє виявити “вузькі місця” в інфраструктурі, які не проявляються під час звичайного навантаження. Наприклад, тест може виявити, що певний брандмауер або база даних стає точкою відмови під час великого напливу трафіку.

За даними досліджень Radware, саме мережевий екран чи міжмережевий екран часто є причиною збою під час DDoS, якщо його продуктивність виявилась нижчою, ніж обсяг атаки.[26]

Регулярні тестування допомагають вчасно знайти такі слабкі ланки і усунути їх (масштабувати або налаштувати по-іншому) до того, як ними скористаються зловмисники. По-друге, симуляція атаки перевіряє ефективність систем виявлення і захисту. Можна побачити, чи спрацювали правила WAF, чи сповіщення CloudWatch були відправлені команді вчасно, чи алгоритми Shield Advanced правильно ідентифікували трафік як атаку. Якщо щось з цього не відбулося, це сигнал допрацювати політики безпеки (наприклад, додати недостаючі сигнатури до WAF, налаштувати пороги алертів і т.д.). По-третє, такі тести перевіряють готовність команди реагування і процедур аварійного відновлення. Під час симуляції персонал (адміністратори, розробники on-call) можуть відпрацювати свої дії: від слідкування за метриками до перемикання трафіку, активації резервних сторінок або взаємодії з підтримкою хмарного провайдера. Це своєрідне навчання на практиці, що значно підвищує навички та впевненість команди у випадку реальної атаки.

Для бізнесу важливим фактором є також дотримання вимог безпеки і стандартів. У деяких галузях (фінанси, телеком тощо) регулятори вимагають проводити стрес-тестування на кіберстійкість, включно з DDoS-атаками, і надавати результати аудиту. Симуляція DDoS в хмарі дозволяє задокументувати, що компанія виконала належні заходи і її план реагування на відмову протестований.

Це підвищує довіру клієнтів і партнерів, адже демонструє проактивний підхід до безпеки. З технічної точки зору, результати таких тестів часто фіксуються у вигляді KPI: відсоток втрати пакетів, середній час відповіді сервера під час атаки, час відновлення після закінчення атаки, максимальне навантаження, яке витримала система тощо[27].

Ці метрики допомагають кількісно оцінити рівень стійкості та відмовостійкості інфраструктури. Варто підкреслити, що всі симуляції повинні проводитись обережно і у тісній взаємодії з провайдером хмари. Незаконно організована атака навіть “на себе” може бути сприйнята як справжня і

порушити роботу сусідніх систем. Тому AWS вимагає використовувати лише авторизовані інструменти для DDoS-тестування і заздалегідь повідомляти про плани таких випробувань.

BreakingPoint Cloud та деякі інші сервіси (MazeBolt, Red Button тощо) схвалені AWS і Azure як офіційні партнери для проведення безпечних атак у хмарі.

Це означає, що трафік атаки буде генеруватися з середовища, яке контролюється та не завдасть побічної шкоди іншим клієнтам. Таким чином, підприємства можуть бути впевнені, що їхні тести відповідають правилам і не призведуть до відключення їхніх облікових записів. Загалом, впровадження регулярних тестів на DDoS-стійкість — важлива складова розробки та експлуатації сучасних веб-систем. За даними компанії Keysight (Ixia), комплексне тестування із застосуванням реального трафіку та атак здатне на 80% знизити ризик деградації мережі під час інцидентів і майже на 70% підвищити готовність IT-інфраструктури до кібератак.

Іншими словами, інвестуючи час у симуляцію найгірших сценаріїв, компанія виграє у надійності: система працюватиме безвідмовно навіть у ворожому середовищі, а команда матиме чіткий план дій. Для розробників та архітекторів такі тести дають зворотний зв’язок щодо проектних рішень — чи достатньо масштабована архітектура, чи правильно розставлені мережеві межі безпеки, чи витримає база даних сплеск коннектів. У разі виявлення проблем вони можуть вдосконалити систему (наприклад, додати шар кешування, оптимізувати алгоритми обробки черг запитів, збільшити ліміти підписки Shield Advanced тощо). Підсумовуючи, розглянутий теоретичний підхід до тестування стійкості до DDoS-атак демонструє, як важливо поєднувати технологічні засоби (симулятори трафіку, хмарні служби безпеки) з організаційними заходами (планування, аналіз результатів, навчання персоналу). Лише така всебічна підготовка забезпечить дійсно високу готовність веб-ресурсу, розгорнутого у хмарі AWS, протистояти сучасним багатовекторним DDoS-атакам без втрати

доступності для користувачів.

4.3 Перспективи розвитку веб ресурсу

На поточному етапі розробки вебресурс ресторанного меню являє собою статичний сайт, реалізований засобами HTML, CSS та JavaScript. Це означає, що сторінки підготовлені заздалегідь і видаються клієнту без серверної обробки — усі користувачі бачать однаковий вміст, який зберігається у вигляді готових файлів на сервері. Такий статичний підхід забезпечує простоту розгортання та швидке завантаження сторінок завдяки відсутності звернень до баз даних чи динамічного генерування контенту на сервері.[28]

Водночас статичний вебсайт обмежений у функціональності: немає можливості інтерактивної взаємодії з користувачем або автоматичного оновлення даних у реальному часі. У перспективі вебресурс може бути модернізований до повноцінного динамічного вебзастосунку. Перехід до динамічної архітектури передбачає запровадження серверної логіки, наприклад на основі Node.js — високопродуктивного середовища виконання JavaScript, спроєктованого для побудови масштабованих мережевих застосунків. На відміну від статичного сайту, динамічний застосунок генерує сторінки «на льоту» відповідно до запитів користувачів, використовуючи серверні скрипти та бази даних для формування персоналізованого контенту.

Зокрема, Node.js у поєднанні з фреймворком (наприклад, Express) може реалізувати бекенд-логіку для обробки запитів — відображення актуального меню, надсилання форм бронювання тощо — тоді як дані меню та бронювань зберігатимуться у системі керування базами даних. Використання бази даних розширить можливості вебресурсу: з’явиться підтримка зберігання та оновлення інформації про страви, замовлення чи резервування столиків у персистентному сховищі.[29] В якості рішення можуть бути розглянуті Amazon RDS або Amazon DynamoDB — хмарні сервіси зберігання даних на платформі AWS. Amazon RDS (Relational Database Service) надає повністю керований

реляційний сервіс, що спрощує налаштування, експлуатацію та масштабування реляційних СУБД в хмарі AWS.

Це означає, що розробник може обрати знайомий рушій (MySQL, PostgreSQL, тощо) і довірити AWS автоматичне управління низькорівневими задачами (резервне копіювання, патчі, реплікація) при збереженні можливості вертикального масштабування бази даних під збільшення навантаження. Альтернативно, Amazon DynamoDB — це повністю керована NoSQL база даних (ключ-значення), яка не потребує адміністрування серверів і забезпечує стабільно малий час відгуку при будь-якому масштабі навантаження.

DynamoDB є серверлес-рішенням, отже автоматично розподіляє дані і навантаження, дозволяючи застосунку прозоро масштабуватися під зростання кількості користувачів чи операцій. Вибір між реляційною СУБД (Amazon RDS) та NoSQL (DynamoDB) залежатиме від характеру даних ресторанного меню та функціональних вимог: реляційна модель підходить для структурованих даних і складних запитів (наприклад, таблиці замовлень, меню з категоріями), тоді як DynamoDB може ефективно обслуговувати дуже високі навантаження на прості операції читання/запису (наприклад, лічильники переглядів страв, кешування доступності столиків тощо).

У подальшому, після інтеграції серверної частини та бази даних, можна реалізувати API для взаємодії з зовнішніми сервісами або клієнтськими додатками. Наприклад, відкритий API може забезпечити можливість стороннім програмам отримувати актуальне меню, а також дозволить впровадити функціонал онлайн-бронювання столиків через сторонні системи чи власний мобільний додаток.[30] Такі API-інтеграції підвищать зручність для користувачів (онлайн-резервування у реальному часі) і гнучкість управління контентом (оновлення меню менеджерами ресторану через захищений інтерфейс). Із зростанням складності та відвідуваності вебресурсу виникає потреба у масштабуванні обчислювальної інфраструктури. Наразі застосунок розгорнуто, ймовірно, на віртуальному сервері EC2 типу t3.micro — це базовий

інстанс загального призначення з 2 vCPU (віртуальних процесорів) та 1 ГБ ОЗП, який належить до сімейства burstable (з можливістю нетривалого прискорення продуктивності).

Така конфігурація підходить для початкового етапу або невеликого навантаження, коли вебресурс обслуговує обмежену кількість користувачів і переважно статичний вміст. Однак після модернізації до динамічного застосунку і появи більш інтенсивних завдань (обробка користувацьких запитів, звернення до бази даних, виконання бізнес-логіки) t3.micro може стати недостатнім. AWS забезпечує гнучкість вертикального масштабування — зміну типу EC2-інстансу на потужніший — без необхідності значної перебудови застосунку.[31] В рамках тієї ж родини T3 доступні більші конфігурації, такі як t3.medium та t3.large, які відрізняються передусім обсягом оперативної пам’яті та допустимим постійним навантаженням на процесор. Наприклад, інстанс t3.medium також має 2 vCPU, проте вже 4 ГБ RAM, а t3.large — 2 vCPU та 8 ГБ RAM, що дозволяє обробляти більше одночасних запитів і кешувати більше даних у пам’яті. Збільшення обсягу пам’яті особливо актуальне при розгортанні серверної платформи Node.js та СУБД: більше RAM дає змогу тримати в пам’яті результати частих запитів, пули підключень до бази даних та інші ресурсоємні структури, зменшуючи затримки вводу-виводу.

Окрім того, більші T3-інстанси мають вищий базовий ліміт CPU-кредитів, що означає здатність довше підтримувати високе навантаження на процесор без зниження продуктивності. Критеріями для переходу на вищий тариф можуть бути сталі показники моніторингу, такі як завантаженість CPU понад ~70-80% або споживання пам’яті, близьке до межі наявної у t3.micro. У разі коли вебзастосунок почне регулярно виходити за межі можливостей t3.micro (наприклад, збільшиться трафік користувачів, додадуться складні операції на сервері, інтеграція з зовнішніми API тощо), доцільно масштабуватися до t3.medium. Цей крок забезпечить запас ресурсів для стабільної роботи: за даними AWS, on-demand вартість t3.medium становить

близько $0.0416 за годину (проти ~$0.0104/год для t3.micro), тобто приблизно $30 на місяць при неперервній роботі.[32]

Подальше зростання аудиторії або функціональності, яке призведе до високого навантаження навіть на t3.medium (наприклад, одночасне перебування великої кількості користувачів, інтенсивне використання бази даних, обробка зображень меню тощо), може вимагати переходу на t3.large. T3.large надає 8 ГБ оперативної пам’яті, що вдвічі більше за t3.medium, і дозволяє значно підвищити обсяг даних, з якими застосунок може працювати в пам’яті (наприклад, тримати більше елементів меню або сесій користувачів у кеші). Вартість t3.large при цьому зростає до ~$0.0832 за годину (близько $60-65 на місяць). У таблиці 4.3.1 нижче підсумовано ключові характеристики зазначених типів інстансів EC2 серії T3.

Таблиця 4.1 — Порівняння конфікураці інстансів

Тип інстансу EC2	Кількість vCPU	Оперативна пам’ять	Тип сховища	Вартість
T3.micro	2	1	EBS (SSD, gp)	$
T3.medium	2	4	EBS (SSD, gp)	$
T3.large	2	8	EBS (SSD, gp)	$

Ще одним перспективним напрямом розвитку інфраструктури є перехід до контейнеризованого середовища розгортання. Контейнеризація передбачає упакування вебзастосунку та його залежностей у легковагові контейнери (наприклад, Docker), що дозволяє запускати їх на спільній інфраструктурі ізольовано один від одного.[33] На платформі AWS для роботи з контейнерами існують керовані сервіси оркестрації — Amazon ECS (Elastic Container Service) та Amazon EKS (Elastic Kubernetes Service). Використання таких сервісів у

перспективі здатне підвищити гнучкість і масштабованість розгортання вебресурсу. Зокрема, Amazon ECS є повністю керованим оркестратором контейнерів, який значно спрощує процес деплойменту, управління та автоматичного масштабування контейнеризованих застосунків. За допомогою ECS можна визначити завдання і служби, налаштувати балансування навантаження між контейнерами і забезпечити швидке масштабування шляхом додавання нових екземплярів контейнерів під збільшення трафіку. Amazon EKS, у свою чергу, надає керований Kubernetes у хмарі AWS, що дає максимальну гнучкість у розгортанні і управлінні контейнерами завдяки можливостям екосистеми Kubernetes. EKS забезпечує високодоступний контрольний кластер Kubernetes і дозволяє використовувати стандартні Kubernetes-інструменти для оркестрації, що особливо корисно при переході до мікросервісної архітектури. У майбутньому перенесення ресторанного вебзастосунку в контейнерну інфраструктуру може означати, що кожен компонент (наприклад, фронтенд, бекенд Node.js, база даних) буде запускатися у власному контейнері. Це відкриває шлях до гнучкого горизонтального масштабування — наприклад, автоматичного додавання копій контейнерів вебсерверу під час пікових навантажень і видалення їх при спаді трафіку. Крім того, AWS дозволяє запускати контейнери не тільки на виділених EC2-вузлах, але й в режимі Fargate (безсерверне виконання), коли ресурси під контейнер виділяються автоматично без управління серверами. Подібний підхід ще більше спрощує експлуатацію: розробники концентруються на коді, а не на підтримці серверів.[34] В результаті впровадження контейнеризації та оркестрації в AWS (через ECS/EKS) вебресурс отримує низку переваг: підвищену відмовостійкість (контейнери можуть розгортатися на кількох вузлах і зонах доступності), легкість розгортання оновлень (заміна версії образу і поступове оновлення контейнерів без простою), а також більш ефективне використання ресурсів (кілька контейнерів можуть працювати на одному EC2-хості). Таким чином, у довгостроковій перспективі перехід від монолітного розгортання на одному

віртуальному сервері до хмарного контейнерного середовища здатен забезпечити масштабування вебзастосунку «меню ресторану» до рівня виробничого навантаження, зберігаючи при цьому гнучкість і керованість інфраструктури на платформі AWS. Висновок: Розвиток обчислювальної інфраструктури вебресурсу на AWS повинен відбуватися поступово, у відповідності до зростання потреб застосунку. Статичний сайт може еволюціонувати в динамічний вебзастосунок на Node.js з інтегрованою базою даних для забезпечення інтерактивності та автоматизації оновлень контенту. При збільшенні відвідуваності та навантаження передбачене гнучке масштабування — спочатку вертикальне (перехід на потужніші EC2-інстанси t3.medium, t3.large), а в подальшому — горизонтальне у контейнеризованому середовищі (кластер AWS ECS/EKS), що гарантуватиме стабільну роботу сервісу під будь-яким навантаженням. Такий підхід відповідає сучасним практикам розробки хмарних застосунків і забезпечує запас міцності для майбутнього розвитку вебресурсу.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі виконано дослідження та практичну реалізацію обчислювальної інфраструктури розгортання вебресурсу на базі хмарної платформи Amazon Web Services. Проведено огляд і аналіз сучасних хмарних технологій, розглянуто концепції IaaS, моделі масштабування, безпеки та відмовостійкості, а також детально проаналізовано можливості AWS як провідного провайдера хмарних обчислень.

У процесі роботи було обґрунтовано вибір AWS як платформи для реалізації інфраструктури розгортання вебдодатків, досліджено її глобальну інфраструктуру, основні сервіси (EC2, S3, IAM, VPC, Route 53, CloudWatch) та принципи взаємодії між ними. Розглянуто моделі тарифікації ресурсів AWS та шляхи оптимізації витрат. Здійснено поетапне проєктування віртуального середовища, що включає створення приватної мережі, налаштування таблиць маршрутизації, конфігурацію правил безпеки та моніторинг стану інфраструктури.

Практична частина роботи присвячена розгортанню реального вебзастосунку — персонального портфоліо, написаного на платформі Node.js. Здійснено запуск EC2-інстанса, налаштовано мережеве середовище за допомогою Amazon VPC, підключено EBS-томи та налаштовано шифрування з використанням AWS KMS. Встановлено та налаштовано вебсервер, проведено розгортання коду, організовано зберігання статичних ресурсів у S3, підключено домен через Route 53, реалізовано моніторинг та сповіщення за допомогою CloudWatch. Усі ресурси було взаємопов’язано через політики IAM з урахуванням принципу найменших привілеїв.

В результаті виконано розгортання повнофункціонального вебресурсу з використанням стандартних засобів AWS, що дозволило забезпечити гнучкість, масштабованість, безпечність та централізоване управління інфраструктурою. Розроблена інфраструктура є модульною та придатною до масштабування, що дозволяє легко адаптувати її під інші типи вебзастосунків і

розширювати функціонал у майбутньому. Практична цінність розробки полягає в її придатності для повторного використання як шаблону для створення високонадійних вебресурсів у середовищі AWS, що особливо актуально для малого бізнесу, освітніх проєктів та персональних розробок.

Запропоноване рішення підтверджує ефективність використання хмарної платформи AWS для побудови обчислювальної інфраструктури вебзастосунків, демонструє практичну реалізацію розгортання, підвищення доступності та безпеки сервісів, а також показує доцільність переходу на хмарні моделі обробки даних у сучасних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Amazon Web Services. What is Amazon EC2? (Amazon EC2 User Guide). Доступ за адресою: https://docs.aws.amazon.com/AWSEC2/

2. Amazon Web Services. What is Amazon VPC? (Amazon Virtual Private Cloud User Guide). Доступ за адресою: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

3. Amazon Web Services. What is Amazon S3? (Amazon S3 User Guide). Доступ за адресою: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

4. Amazon Web Services. What is AWS Identity and Access Management (IAM)? (AWS IAM User Guide). Доступ за адресою: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

5. Amazon Web Services. What is Amazon Route 53? (Amazon Route 53 Developer Guide). Доступ за адресою: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html

6. Amazon Web Services. What is Amazon CloudWatch? (Amazon CloudWatch User Guide). Доступ за адресою: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

7. Amazon Web Services. How AWS Pricing Works (Whitepaper). Доступ за адресою: https://docs.aws.amazon.com/whitepapers/latest/how-aws-pricing-works/abstract-and-introduction.html

8. Amazon Web Services. AWS Global Infrastructure. Доступ за адресою: https://aws.amazon.com/about-aws/global-infrastructure/

9. Node.js. Official Node.js Documentation (API Reference). Доступ за адресою: https://nodejs.org/api/

10. Mell, P. & Grance, T. The NIST Definition of Cloud Computing (NIST SP 800-145). Доступ за адресою: https://csrc.nist.gov/publications/detail/sp/800-145/final

11. Softprom. Amazon Simple Storage Service (Amazon S3) — хмарне об'єктне сховище. Доступ за адресою: https://softprom.com/ua/vendor/amazon-web-services/product/amazon-simple-storage-service-amazon-s3

12. Альвіна. Покроковий гід по налаштуванню VPS для ефективної роботи з AWS. Доступ за адресою: https://rx-name.ua/blog/pokrokovyj-gid-po-nalashtuvannyu-vps-dlya-efektyvnoyi-roboty-z-amazon-web-services-aws

13. Хмарні технології AWS для українського бізнесу. Доступ за адресою: https://itedu.center/ua/blog/sysadministration/khmarni-tekhnolohii-aws-dlia-ukrainskoho-biznesu/

14. Набатова, Алла. Навіщо бізнес-аналітику сертифікація AWS Cloud Practitioner. Доступ за адресою: https://www.ba.in.ua/2021/06/27/navishho-biznes-analityku-sertyfikacziya-aws-cloud-practitioner/

15. EPAM University. Як стати AWS Cloud Engineer? Доступ за адресою: https://campus.epam.ua/ua/skill/AWS

16. Компанієць, Федір. 20 способів оптимізації витрат на AWS, які легко допоможуть заощадити 80+% бюджету. Доступ за адресою: https://dou.ua/forums/topic/46648/

17. IT Edu Center. AWS: цінність клауду в сучасному IT та чому AWS витрачає великі кошти на створення дата-центрів. Доступ за адресою: https://itedu.center/ua/blog/devops/aws-cinnist-klaudu-v-suchasnomu-it/

18. Хмара TechExpert. Що таке хмарні обчислення: використання, переваги та типи хмарних обчислень. Доступ за адресою: https://onbiz.biz/cloud-computing-possibilities/

19. Amazon Web Services, Inc. Elastic Compute Cloud (EC2) Instance Types. — Режим доступу: https://aws.amazon.com/ec2/instance-types/

20. AWS Pricing Calculator — EC2. — Режим доступу: https://calculator.aws.amazon.com

21. AWS Documentation — Amazon EC2 T3 Instances. — Режим доступу: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t3-instances.html

22. Node.js Foundation. Node.js Documentation. — Режим доступу: https://nodejs.org/en/docs/

23. Express.js — Fast, unopinionated, minimalist web framework for Node.js. — Режим доступу: https://expressjs.com/

24. GitHub Docs — Cloning a repository. — Режим доступу: https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository

25. Git Documentation. — Режим доступу: https://git-scm.com/docs

26. Mozilla Developer Network. Static— Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Pages_static_or_dynamic

27. Amazon Web Services. Amazon RDS Documentation. — Режим доступу: https://docs.aws.amazon.com/rds/

28. Amazon Web Services. Amazon DynamoDB Documentation. — Режим доступу: https://docs.aws.amazon.com/amazondynamodb/

29. npm Docs. npm install. — Режим доступу: https://docs.npmjs.com/cli/v9/commands/npm-install

30. Docker Documentation. — Режим доступу: https://docs.docker.com/

31. Amazon Web Services. ECS vs EKS Overview. — Режим доступу: https://aws.amazon.com/containers/

32. AWS EC2 On-Demand Pricing (t3.micro, t3.medium, t3.large). — Режим доступу: https://aws.amazon.com/ec2/pricing/on-demand/

33. Amazon Cloud Architecture Center — Best practices. — Режим доступу: https://aws.amazon.com/architecture/

34. Amazon Elastic Block Store (EBS) — Документація. — Режим доступу: https://docs.aws.amazon.com/ebs/

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О.Д.

«21» березня 2025 року

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

««Обчислювальна інфраструктура розгортання вебресурсу на хмарній платформі AWS»»

08-54.БКР.005.00.000 ТЗ

Науковий керівник к.т.н. доц. каф. ОТ

_____ Кадук О.В.

Студент групи 1КІ-21б

_____ Глоба Н.С.

Вінниця 2025

1 Підстави для виконання бакалаврської кваліфікаційної роботи (БКР)

1.1 Розробка системи аналізу та оптимізації продуктивності веб-додатку є актуальною через зростання потреби у високопродуктивних та надійних веб—сервісах, а також через збільшення частоти кібератак, таких як DDoS, що ставлять під загрозу доступність онлайн-систем. Сучасні виклики, як-от раптові пікові навантаження, складність розподілених архітектур та необхідність обробки великої кількості одночасних користувачів, вимагають створення ефективних методик тестування і оптимізації продуктивності. Забезпечення стійкості веб-додатків до інтенсивного трафіку та зловмисних атак сприяє підвищенню стабільності та якості обслуговування користувачів. Інтеграція таких підходів у процес розробки (DevOps) та використання хмарних інфраструктур дозволяє своєчасно виявляти й усувати «вузькі місця» у продуктивності.

1.2 Наказ про затвердження теми БКР ректором університету № 112 від 10 квітня 2025 року. 2 Мета БКР і призначення розробки

2 Мета КБКР і призначення розробки

2.1 Метою роботи є підвищення продуктивності та стійкості веб-додатку шляхом розробки системи для комплексного тестування навантаження та оптимізації, яка забезпечить надійний аналіз показників роботи додатку під високим навантаженням і в умовах мережевих атак. Така система дозволить виявити та усунути «вузькі місця» в архітектурі додатку, забезпечуючи його стабільну роботу й масштабованість.

2.2 Призначення розробки полягає у створенні програмної системи, здатної моделювати високі навантаження і атаки на веб-додаток, збирати та аналізувати метрики продуктивності і надавати рекомендації для оптимізації. Для цього використовуються сучасні підходи, такі як автоматизоване навантажувальне тестування (генерація великої кількості запитів) та хмарні сервіси симуляції DDoS-атак. Таким чином, розроблена система допоможе

забезпечити надійну роботу веб-додатку під час пікового трафіку та зменшити вплив зловмисних впливів, підвищуючи якість сервісу та довіру користувачів.

3 Вихідні дані для виконання БКР

3.1 Проведення аналізу сучасних методів навантажувального тестування та оптимізації веб-додатків, а також огляд інструментів і бібліотек, що використовуються у сфері тестування продуктивності веб-систем у середовищі Node.js та хмарних платформ.

3.2 Проєктування архітектури системи та розробка загальної структури, визначення основних модулів (модуль генерації навантаження, моніторингу та збору метрик, модуль аналізу результатів, інтеграції з DDoS-симулятором тощо) та принципів їх взаємодії.

3.3 Реалізація функціональних компонентів та розробка комп’ютерної моделі роботи системи: створення модулів генерації трафіку для навантажувального тестування, збору та обробки показників продуктивності, симуляції DDoS-атак і аналізу отриманих результатів з метою подальшої оптимізації веб-додатку.

3.4 Тестування та перевірка працездатності — визначення тестових сценаріїв (різні рівні інтенсивності навантаження, моделювання атак) для перевірки правильності оцінювання продуктивності та стійкості веб-додатку; аналіз отриманих даних для визначення, чи відповідають показники продуктивності заданим критеріям.

3.5 Відповідність вимогам безпеки та забезпечення сумісності із обраною інфраструктурою (локальним середовищем та хмарною платформою) та дотримання принципів безпечного проведення навантажувального тестування, щоб уникнути негативного впливу на реальні системи.

4 Вимоги до виконання БКР

Головна вимога — реалізувати систему для комплексного тестування та оптимізації продуктивності веб-додатку, яка забезпечує надійне виявлення

вузьких місць та оцінку продуктивності під час високих навантажень і мережевих атак, з метою підвищення надійності та масштабованості веб-сервісу.

5 Етапи БКР та очікувані результати приведені у таблиці А.1.

Таблиця А.1 — Етапи виконання роботи

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Постановка задачі роботи. Вступ	21.03.25	23.03.25	Вступ
2	Аналіз методів та способів виділення та розпізнавання рукописних символів	24.03.25	30.03.25	розділ 1
3	Аналіз ринку, обираннння технології розробку	01.04.25	21.04.25	Розділ 2
4	Створення хмарного додаткого на платформі AWS	22.04.25	05.05.25	Розділ 3, розробка програми
5	Тестування веб додатку, аналіз можливостей покращення	06.05.25	10.05.25	Розділ 4
6	Оформлення пояснювальної записки та ілюстративного матеріалу	11.05.25	12.05.25	ПЗ, презентація
7	Перевірка якості виконання бакалаврської роботи та усунення недоліків	13.05.25	13.05.25	ПЗ, презентація

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, графічні та ілюстративні матеріали, протокол попереднього захисту на кафедрі, відгуки наукового керівника та опонента, протоколи складання державних екзаменів, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія» (освітня програма «Системне програмування»). Кафедра обчислювальної техніки ВНТУ 2023.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21».

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: _ Обчислювальна інфраструктура розгортання вебресурсу на хмарній платформі AWS _

Тип роботи: _____ бакалаврська кваліфікаційна робота _____

(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ _____кафедра обчислювальної техніки_____

(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism __1___%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✔ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

- У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

- У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О.Д., зав. каф. ОТ _____ _____

(прізвище, ініціали, посада) (підпис)

____Тарновський М Г., доц. каф ОТ_____ _____

(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ ___Захарченко С.М.____

(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ _____Кадук О.В._____

(підпис) (прізвище, ініціали, посада)

Здобувач _____ _____Глоба Н.С._____

(підпис) (прізвище, ініціали)

ДОДАТОК В

Графічна частина

ОБЧИСЛЮВАЛЬНА ІНФРАСТРУКТУРА РОЗГОРТАННЯ ВЕБРЕСУРСУ НА ХМАРНІЙ ПЛАТФОРМІ AWS

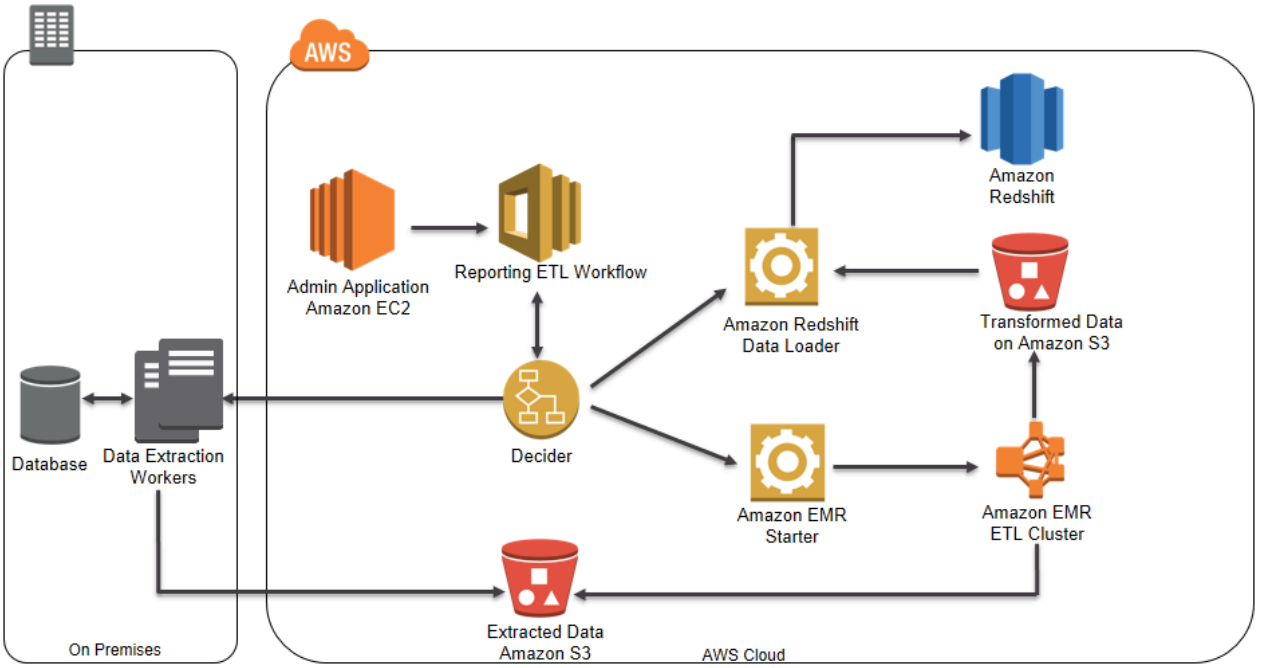

Рисунок В.1 — Алгоритм роботи AWS сервісів

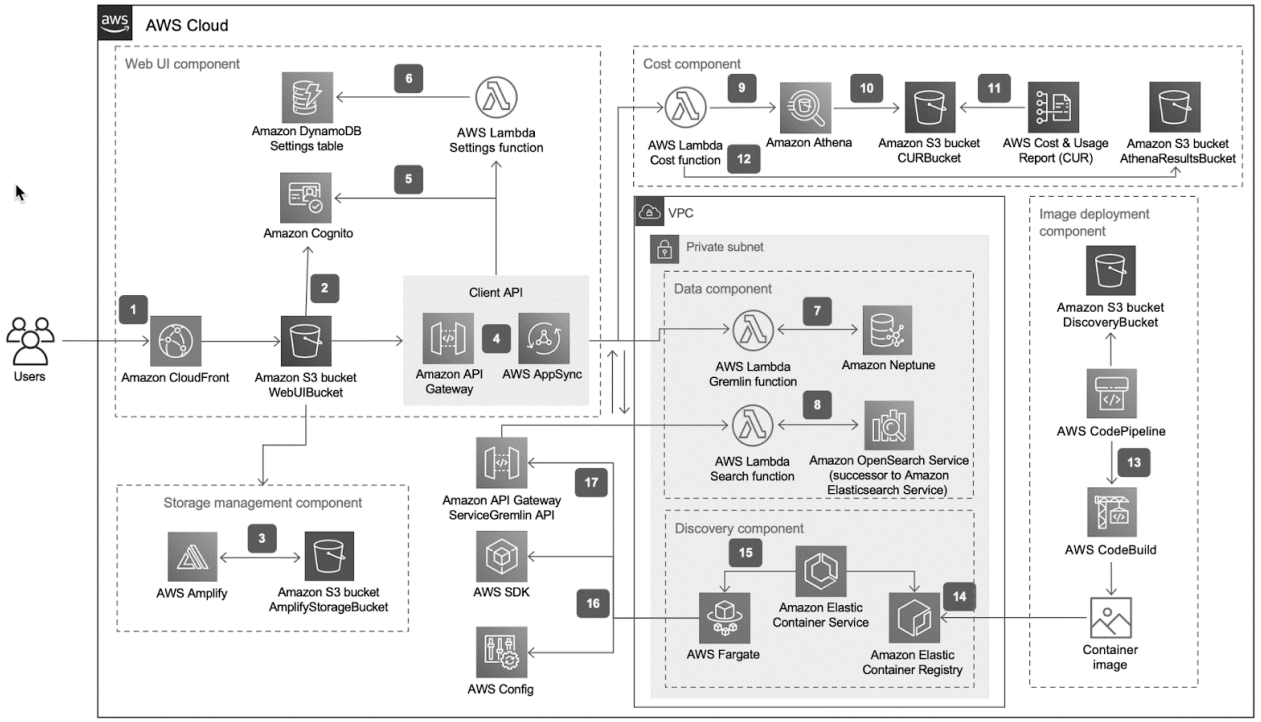

Рисунок В.2 — Структурна візуалізація середовища AWS

ДОДАТОК Г
Ілюстративна частина

ОБЧИСЛЮВАЛЬНА ІНФРАСТРУКТУРА РОЗГОРТАННЯ ВЕБРЕСУРСУ НА ХМАРНІЙ ПЛАТФОРМІ AWS

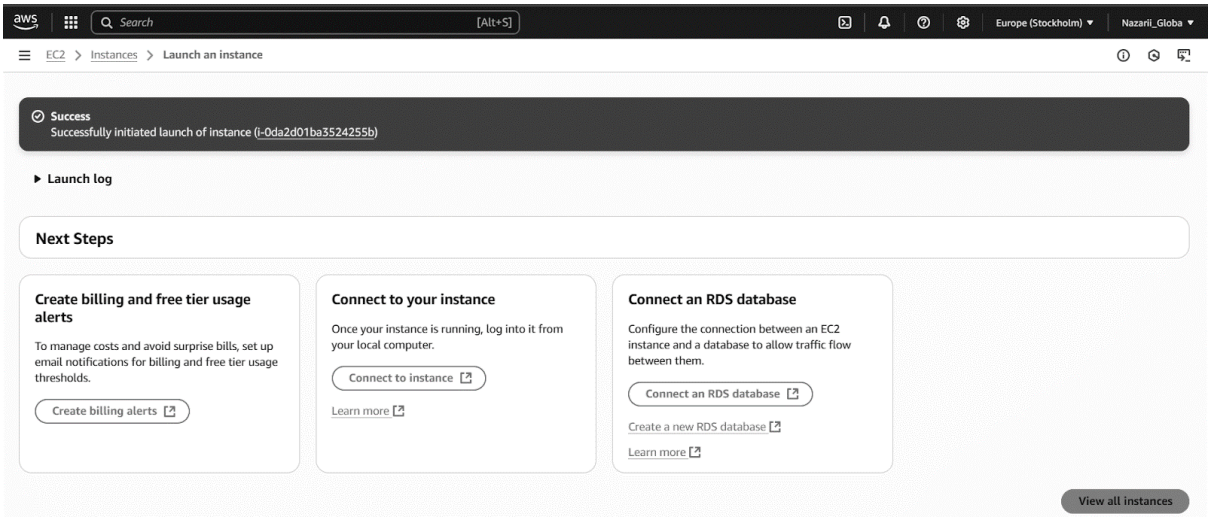

Рисунок Г.1 — повідомлення про успішний запуск інстансу

ДОДАТОК Д

Лістинг програми

```
//controller.js

import * as model from './model.js';
import {MODAL_CLOSE_SEC} from './config.js';
import recipeView  from './views/recipeView.js';
import searchView  from './views/searchView.js';
import resultsView from './views/resultsView.js';
import bookmarksView from './views/bookmarksView.js';
import paginationView from './views/paginationView.js';
import AddRecipeView from './views/addRecipeView.js';

import 'core-js/stable';
import 'regenerator-runtime/runtime';
import addRecipeView from './views/addRecipeView.js';

if(module.hot) {
  module.hot.accept();
}

const recipeContainer = document.querySelector('.recipe');

const controlRecipes = async function() {
  try {
    const id = window.location.hash.slice(1);
    console.log(id);

    if (!id) return;
    recipeView.renderSpinner();

    // 0) Update results view to mak selected search result
    bookmarksView.update(model.state.bookmarks);

    // 1) Updating bookmarks view
    resultsView.update(model.getSearchResultsPage());

    // 2) Loading recipe
    await model.loadRecipe(id);

    // 3) Rendering recipe
    recipeView.render(model.state.recipe);
```

```
  } catch (err) {
    recipeView.renderError();
    console.error(err);
  }
};

const controlSearchResults = async function() {
  try {
    resultsView.renderSpinner();

    // 1) Get search query
    const query = searchView.getQuery();
    if ( !query ) return;

    // 2) Load search results
    await model.loadSearchResults(query);

    // 3) Render results
    //resultsView.render(model.state.search.results);
    resultsView.render(model.getSearchResultsPage());

    // 4) Render initial pagination buttons
    paginationView.render(model.state.search);
  } catch (err) {
    console.log(err);
  }
}

const controlPagination = function(goToPage) {
  // 1) Render NEW results
  //resultsView.render(model.state.search.results);
  resultsView.render(model.getSearchResultsPage(goToPage));

  // 2) Render NEW initial pagination buttons
  paginationView.render(model.state.search);
};

const controlServings = function(newServings) {
  // Update the recipe servings (in state)
  model.updateServings(newServings);

  // Update the recipe view
  //recipeView.render(model.state.recipe);
  recipeView.update(model.state.recipe);
```

```
};

const controlAddBookmark = function() {
  // 1) Add/remove bookmark
  if(!model.state.recipe.bookmarked) model.addBookmark(model.state.recipe);
  else model.deleteBookmark(model.state.recipe.id);

  // 2) Update recipe view
  recipeView.update(model.state.recipe);

  // 3) Render bookmarks
  bookmarksView.render(model.state.bookmarks);
};

const controlBookmarks = function() {
  bookmarksView.render(model.state.bookmarks);
}

const controlAddRecipe = async function(newRecipe) {
  try {
    // Show loading spinner
    addRecipeView.renderSpinner();

    // Upload the new recipe data
    await model.uploadRecipe(newRecipe);
    console.log(model.state.recipe);

    // Render recipe
    recipeView.render(model.state.recipe);

    // Success message
    addRecipeView.renderMessage();

    // Render bookmark view
    bookmarksView.render(model.state.bookmarks);

    // Change ID in URL
    window.history.pushState(null, '', `#${model.state.recipe.id}`);

    // Close form window
    setTimeout(function() {
      addRecipeView.toggleWindow();
    }, MODAL_CLOSE_SEC * 1000);
```

```
  } catch(err) {
    console.log('', err);
    addRecipeView.renderError(err.message);
  }
};

const init = function() {
  bookmarksView.addHandlerRender(controlBookmarks);
  recipeView.addHandlerRender(controlRecipes);
  recipeView.addHandlerUpdateServings(controlServings);
  recipeView.addHandlerAddBookmark(controlAddBookmark);
  searchView.addHandlerSearch(controlSearchResults);
  paginationView.addHandlerClick(controlPagination);
  addRecipeView.addHandlerUpload(controlAddRecipe);
};

init();

// model.js
import { API_URL, RES_PER_PAGE, KEY } from "./config.js";
import { AJAX } from "./helpers.js";

export const state = {
    recipe: {},
    search: {
        query: '',
        results: [],
        page: 1,
        resultsPerPage: RES_PER_PAGE,
    },
    bookmarks: [],
};

const createRecipeObject = function(data) {
    const {recipe} = data.data;
    return {
        id: recipe.id,
        title: recipe.title,
        publisher: recipe.publisher,
        sourceUrl: recipe.source_url,
        image: recipe.image_url,
        servings: recipe.servings,
        cookingTime: recipe.cooking_time,
        ingredients: recipe.ingredients,
```

```
        ...(recipe.key && {key: recipe.key}),
    };
}

export const loadRecipe = async function(id) {
    try {
        const data = await AJAX(`${API_URL}/${id}?key=${KEY}`);
        state.recipe = createRecipeObject(data);

        if(state.bookmarks.some(bookmark => bookmark.id === id))
            state.recipe.bookmarked = true;
        else
            state.recipe.bookmarked = false;

        console.log(state.recipe);
    } catch (err) {
        // Temp error handling
        console.error(`${err} `);
        throw err;
    }
};

export const loadSearchResults = async function(query) {
    try {
        state.search.query = query;

        const data = await AJAX(`${API_URL}?search=${query}&key=${KEY}`);
        console.log(data);

        state.search.results = data.data.recipes.map(rec => {
            return {
                id: rec.id,
                title: rec.title,
                publisher: rec.publisher,
                image: rec.image_url,
                ...(rec.key && {key: rec.key})
            }
        });
        state.search.page = 1;
    } catch (err) {
        console.error(`${err} `);
        throw err;
    }
};
```

```
export const getSearchResultsPage = function(page = state.search.page) {
    state.search.page = page;
    const start = (page - 1) * state.search.resultsPerPage // 0;
    const end = page * state.search.resultsPerPage // 9;

    return state.search.results.slice(start, end);
};


export const updateServings = function(newServings) {
    state.recipe.ingredients.forEach(ing => {
        ing.quantity = ing.quantity * newServings / state.recipe.servings;
    });

    state.recipe.servings = newServings;
};

const persistBookmarks = function() {
    localStorage.setItem('bookmarks', JSON.stringify(state.bookmarks));
};

export const addBookmark = function(recipe) {
    // Add bookmark
    state.bookmarks.push(recipe);

    // Mark current current recipe as bookmarked
    if(recipe.id === state.recipe.id) state.recipe.bookmarked = true;

    persistBookmarks();
};

export const deleteBookmark = function(id) {
    // Delete bookmark
    const index = state.bookmarks.findIndex(el => el.id === id);
    state.bookmarks.splice(index, 1);

    // Mark current current recipe as NOT bookmarked
    if(id === state.recipe.id) state.recipe.bookmarked = false;

    persistBookmarks();
};

const init = function() {
```

```
    const storage = localStorage.getItem('bookmarks');
    if(storage) state.bookmarks = JSON.parse(storage);
};

init();

const clearBookmarks = function() {
    localStorage.clear('bookmarks')
};

export const uploadRecipe = async function(newRecipe) {
    try {
        const ingredients = Object.entries(newRecipe).filter(entry =>
entry[0].startsWith('ingredient') && entry[1] !== '')
            .map(ing => {
                const ingArr = ing[1].split(',').map(el => el.trim());

                if(ingArr.length !== 3) throw new Error(
                    'Wrong ingridients format! Please use the correct format :)'
                );

                const [quantity, unit, description] = ingArr;
                return { quantity: quantity ? +quantity : null, unit, description };
            });

        const recipe = {
            title: newRecipe.title,
            source_url: newRecipe.sourceUrl,
            image_url: newRecipe.image,
            publisher: newRecipe.publisher,
            cooking_time: +newRecipe.cookingTime,
            servings: +newRecipe.servings,
            ingredients,
        };
        const data = await AJAX(`${API_URL}?key=${KEY}`, recipe);
        state.recipe = createRecipeObject(data);
        addBookmark(state.recipe);
    } catch(err) {
        throw err;
    }
}

// config.js
export const API_URL = 'https://forkify-api.herokuapp.com/api/v2/recipes';
```

```
export const TIMEOUT_SEC = 10;
export const RES_PER_PAGE = 10;
export const KEY = 'bc3302b8-bd01-423a-a8b8-0d6730e7a7bd';
export const MODAL_CLOSE_SEC = 2.5;

// helpers.js

import { TIMEOUT_SEC } from "./config.js";

const timeout = function (s) {
   return new Promise(function (_, reject) {
     setTimeout(function () {
       reject(new Error(`Request took too long! Timeout after ${s} second`));
     }, s * 1000);
   });
};

export const AJAX = async function(url, uploadData = undefined) {
  try {
    const fetchPro = uploadData ? fetch(url, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(uploadData),
    }) : fetch(url);

    const res = await Promise.race([fetchPro, timeout(TIMEOUT_SEC)]);
    const data = await res.json();

    if(!res.ok) throw new Error(`${data.message} (${res.status})`);
    return data;
  } catch(err) {
    throw err;
  }
}

// View.js
import icons from 'url:../../img/icons.svg';

export default class View {
   _data;

   /**
```

```
    * Render the received object to the DOM
    * @param {Object | Object[]} data The data to be rendered (e.g. recipe)
    * @param {boolean} [render=true] If false, create murkup string instead of
rendering to the DOM
    * @returns {undefined | string} A markup string is returned if render=false
    * @this {Object} View instance
    * @author Lesia Hloba
    * @todo Finish implementation
    */
   render(data, render = true) {
      if(!data || (Array.isArray(data) && data.length === 0 )) return this.renderError();

      this._data = data;
      const markup = this._generateMarkup();

      if(!render) return markup;

      this._clear();
      this._parentElement.insertAdjacentHTML('afterbegin', markup);
   }

   update(data) {
      this._data = data;
      const newMarkup = this._generateMarkup();

      const newDom =
document.createRange().createContextualFragment(newMarkup);
      const newElements = Array.from(newDom.querySelectorAll('*'));
      const curElements = Array.from(this._parentElement.querySelectorAll('*'));
      console.log(curElements);
      console.log(newElements);

      newElements.forEach((newEl, i) => {
         const curEl = curElements[i];
         console.log(curEl, newEl.isEqualNode(curEl));

         // Updates changes TEXT
         if(!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== ")
{
            //console.log(", newEl.firstChild.nodeValue.trim());
            curEl.textContent = newEl.textContent;
         }

         // Updates changes ATTRIBUTES
```

```
            if(!newEl.isEqualNode(curEl)) {
                Array.from(newEl.attributes).forEach(attr => curEl.setAttribute(attr.name,
attr.value));
            }
        });
    }

    _clear() {
        this._parentElement.innerHTML = '';
    }

    renderSpinner() {
        const markup = `
        <div class="spinner">
          <svg>
            <use href="${icons}#icon-loader"></use>
          </svg>
        </div>
        `;
        this._clear();
        this._parentElement.insertAdjacentHTML('afterbegin', markup);
    };

    renderError(message = this._errorMessage) {
        const markup = `
        <div class="error">
            <div>
            <svg>
                <use href="${icons}#icon-alert-triangle"></use>
            </svg>
            </div>
            <p>${message}</p>
        </div>
        `;
        this._clear();
        this._parentElement.insertAdjacentHTML('afterbegin', markup);
    }

    renderMessage(message = this._message) {
        const markup = `
        <div class="message">
            <div>
            <svg>
                <use href="${icons}#icon-smile"></use>
```

```
            </svg>
            </div>
            <p>${message}</p>
        </div>
        `;
        this._clear();
        this._parentElement.insertAdjacentHTML('afterbegin', markup);
    }
}

// recipeView.js
import View from './View.js';
import icons from 'url:../../img/icons.svg';

class AddRecipeView extends View {
    _parentElement = document.querySelector('.upload');
    _message = 'Recipe was successfully uploaded :)'

    _window = document.querySelector('.add-recipe-window');
    _overlay = document.querySelector('.overlay');
    _btnOpen = document.querySelector('.nav__btn--add-recipe');
    _btnClose = document.querySelector('.btn--close-modal');

    constructor() {
        super();
        this._addHandlerShowWindow();
        this._addHandlerHideWindow();
    }

    toggleWindow() {
        this._overlay.classList.toggle('hidden');
        this._window.classList.toggle('hidden');
    }

    _addHandlerShowWindow() {
        this._btnOpen.addEventListener('click', this.toggleWindow.bind(this));
    }

    _addHandlerHideWindow() {
        this._btnClose.addEventListener('click', this.toggleWindow.bind(this));
        this._overlay.addEventListener('click', this.toggleWindow.bind(this));
    }

    addHandlerUpload(handler) {
```

```
        this._parentElement.addEventListener('submit', function(e) {
            e.preventDefault();
            const dataArr = [...new FormData(this)];
            const data = Object.fromEntries(dataArr);
            handler(data);
        });
    }

    _generateMarkup() {}
}

export default new AddRecipeView();

// resultsView.js
import View from './View.js';
import previewView from './previewView.js';
import icons from 'url:../../img/icons.svg';

class ResultsView extends View {
    _parentElement = document.querySelector('.results');
    _errorMessage = 'No recipes found for your query! Please try again ;)'
    _message = '';

    _generateMarkup() {
        return this._data.map(result => previewView.render(result, false)).join('');
    }
}

export default new ResultsView();

// bookmarksView.js
import View from './View.js';
import previewView from './previewView.js';
import icons from 'url:../../img/icons.svg';

class BookmarksView extends View {
    _parentElement = document.querySelector('.bookmarks__list');
    _errorMessage = 'No bookmarks yet. Find a nice recipe and bookmark it ;)'
    _message = '';

    addHandlerRender(handler) {
        window.addEventListener('load', handler)
    }
```

```
    _generateMarkup() {
        return this._data.map(bookmark => previewView.render(bookmark,
false)).join('');
    }
}

export default new BookmarksView();

// paginationView.js
import View from './View.js';
import icons from 'url:../../img/icons.svg';

class PaginationView extends View {
    _parentElement = document.querySelector('.pagination');

    addHandlerClick(handler) {
        this._parentElement.addEventListener('click', function(e) {
            const btn = e.target.closest('.btn--inline');
            if(!btn) return;

            const goToPage = +btn.dataset.goto;
            handler(goToPage);
        });
    }

    _generateMarkup() {
        const curPage = this._data.page;
        const numPages = Math.ceil(this._data.results.length /
this._data.resultsPerPage);

        // Page 1, and there are other pages
        if(curPage === 1 && numPages > 1) {
            return `
                <button data-goto="${curPage + 1}" class="btn--inline
pagination__btn--next">
                    <span>Page ${curPage + 1}</span>
                    <svg class="search__icon">
                        <use href="${icons}#icon-arrow-right"></use>
                    </svg>
                </button>
            `;
        }

        // Last page
```

```
        if(curPage === numPages && numPages > 1) {
            return `
                <button data-goto="${curPage - 1}" class="btn--inline
pagination__btn--prev">
                    <svg class="search__icon">
                    <use href="${icons}#icon-arrow-left"></use>
                    </svg>
                    <span>Page ${curPage - 1}</span>
                </button>
            `;
        }

        // Other page
        if(curPage < numPages) {
            return `
            <button data-goto="${curPage - 1}" class="btn--inline
pagination__btn--prev">
                <svg class="search__icon">
                <use href="${icons}#icon-arrow-left"></use>
                </svg>
                <span>Page ${curPage - 1}</span>
            </button>
            <button data-goto="${curPage + 1}" class="btn--inline
pagination__btn--next">
                <span>Page ${curPage + 1}</span>
                <svg class="search__icon">
                <use href="${icons}#icon-arrow-right"></use>
                </svg>
            </button>
            `;
        }

        // Page 1, and there are NO other pages
        return '';
    }
}

export default new PaginationView();

// addRecipeView.js
import View from './View.js';
import icons from 'url:../../img/icons.svg';

class AddRecipeView extends View {
```

```
    _parentElement = document.querySelector('.upload');
    _message = 'Recipe was successfully uploaded :)'

    _window = document.querySelector('.add-recipe-window');
    _overlay = document.querySelector('.overlay');
    _btnOpen = document.querySelector('.nav__btn--add-recipe');
    _btnClose = document.querySelector('.btn--close-modal');

    constructor() {
        super();
        this._addHandlerShowWindow();
        this._addHandlerHideWindow();
    }

    toggleWindow() {
        this._overlay.classList.toggle('hidden');
        this._window.classList.toggle('hidden');
    }

    _addHandlerShowWindow() {
        this._btnOpen.addEventListener('click', this.toggleWindow.bind(this));
    }

    _addHandlerHideWindow() {
        this._btnClose.addEventListener('click', this.toggleWindow.bind(this));
        this._overlay.addEventListener('click', this.toggleWindow.bind(this));
    }

    addHandlerUpload(handler) {
        this._parentElement.addEventListener('submit', function(e) {
            e.preventDefault();
            const dataArr = [...new FormData(this)];
            const data = Object.fromEntries(dataArr);
            handler(data);
        });
    }

    _generateMarkup() {}
}

export default new AddRecipeView();

// searchView.js
class SearchView {
```

```
    #parentEl = document.querySelector('.search');

    getQuery() {
        const query = this.#parentEl.querySelector('.search__field').value;
        this.#clearInput();
        return query;
    }

    #clearInput() {
        this.#parentEl.querySelector('.search__field').value = '';
    }

    addHandlerSearch(handler) {
        this.#parentEl.addEventListener('submit', function(e) {
            e.preventDefault();
            handler();
        });
    }
}

export default new SearchView();
```